Chapter 4

Recurrent Neural Networks

Qiang Ji

Outline

- Sequential data and dynamic systems
- Dynamic Systems
- Recurrent NN
  - Structure
  - Learning
  - Issues
- Long Short -Term Memory (LSTM)
- Applications

Sequential Data

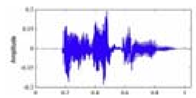
- Also called time series, sequential data is a series of data, resulted from sampling a dynamic process over a discrete of times, i.e., $X_1, X_2, \ldots, X_T$, where X_t is the sample acquired at time t.
- Examples
  - A sequence of image frames
  - A sequence of speech signals
  - A sequence of words
  - A sequence of stock prices

Dynamic Systems

- Input is a time series data $X_1, X_2, \ldots, X_T$
- Output can be another time series $Y_1, Y_2, \ldots, Y_T$ or a class label Y.
- For example
  - Input is a sequence of sound signal and output is the spoken words
  - Input is a sequence of images of body images output is gesture label
  - Input is daily stock prices and output is the prediction of next day or future stock price
  - Input is a sequence of textual words and output is a sentence

Dynamic System Examples

Speech recognition:
Input: Output: recognized speech



Human action recognition
Input: a sequence of body images Output: Jumping

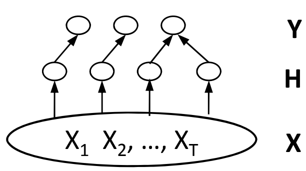
Natural Language Processing:
Input : a sequence of words
walked down the street in a hat with a smile
Output: a sentence

Sequential Data Modeling

- Temporal modeling captures and exploits temporal dependences among all input samples to predict output variables
- Dynamic Models
  - Naïve Temporal Neural Networks
  - Autoregressive model
  - Linear Dynamic Systems
  - Hidden Markov Models
  - Recurrent Neural Networks

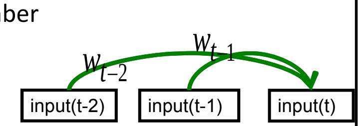
Naive Temporal Neural Network

- Concatenate all data from time 1 to T into one large input vector, i.e., **X**=[X_1, X_2, …, X_T] and use standard NN to represent it
- Issues:
  - How long is T before it becomes too large. Note each X_t is a vector
  - Input sequence varies in T



Autoregressive models

- Predict the next term in a sequence from a fixed number (T) of previous terms



- Issues

$$X[t]=W_{t-1}X[t-1]+W_{t-2}X[t-2]+\ldots+W_{t-T}X[t-T]$$

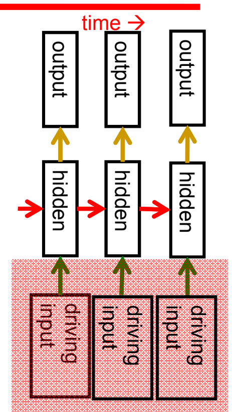
  - ➢ Cannot represent output variables
  - ➢ Assume linear system
  - ➢ Can only go T times back
  - ➢ No memory

Slide from Geoffrey Hinton

Linear Dynamical Systems (LDS)

(engineers love them!)

- These are generative models. They have a real-valued hidden state that cannot be observed directly.
  - Both state transition and output/state are linear with Gaussian noise.
  - There may also be driving inputs.
- To predict the next output (so that we can shoot down the missile) we need to infer the hidden state.
  - A linearly transformed Gaussian is a Gaussian. So the distribution over the hidden state given the data so far is Gaussian. It can be computed using “Kalman filtering”.



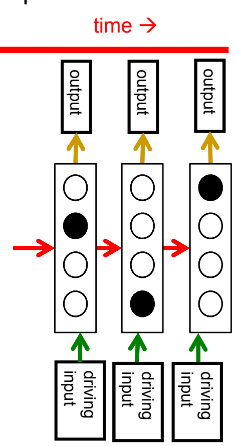
Slide from Geoffrey Hinton

Limitations with LDS

- System is linear
- The hidden state is a single scalar with limited memory
- Assumption of Gaussian noise

Hidden Markov Models (computer scientists love them!)

- Hidden Markov Models have a discrete one-of-N hidden state. Transitions between states are stochastic and controlled by a transition matrix. The outputs produced by a state are stochastic.
  - Both state transition and output are linear and Gaussain
- Because of special topology, HMMs have efficient algorithms for inference and learning.



Slide from Geoffrey Hinton

Limitation with HMMs

- Each time, one of N states can be remembered.
- With N hidden states, it can only remember log(N) bits of information, which is far from enough to store much information

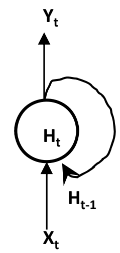
  For example, to remember a sentence, we could need 100 bit, which translates to 2^{100} states, too big!
- Both state transition and state/output are linear with Gaussian noise

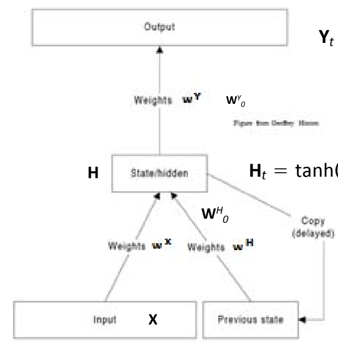
Slide from Geoffrey Hinton

Recurrent Neural Networks

- Like HMM and LDS, RNN has memory. Its memory is represented by a hidden vector of real or binary numbers. The memory can therefore efficiently store a lot of information about the past.
- Unlike HMM or LDS,
  - the transition between states is non-linear
  - It is deterministic –efficient learning and inference
  - It is discriminative –better for classification and regression tasks

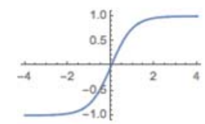
Recurrent Neural Networks





$$\mathbf{Y}_t = \mathrm{softmax}(\mathbf{W}^Y\mathbf{H}_t + \mathbf{W}^Y{}_0)$$

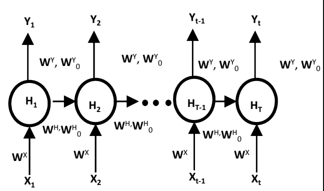
$$\mathbf{H}_t = \tanh(\mathbf{W}^X\mathbf{X}_t + \mathbf{W}^H\mathbf{H}_{t-1} + \mathbf{W}^H{}_0)$$



tanh()

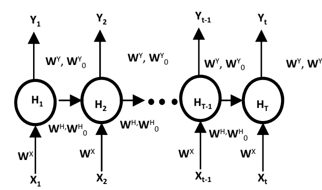
A Vanilla RNN

- Unrolled RNNs over T time slices
- It applies to input sequences of different lengths and predict sequences of different lengths
- Each hidden node summarizes an arbitrary length sequence $(\mathbf{X}_1, \ldots, \mathbf{X}_{T-1})$ to a fixed length vector $\mathbf{H}_T$.
- The weights $\mathbf{W}^H$, $\mathbf{W}^X$, and $\mathbf{W}^Y$ are shared over time



RNN Training

- Treat unrolled RNN as an expanded NN structure over T time slices, with shared weights



- Training with backpropgation through time

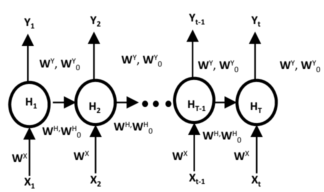
BPTT – Backprop Through Time

- BPTT allows us to look back further as we train
- However we have to pre-specify a value *T*, which is the maximum that learning will look back
- During training we *unfold* the network in time as if it were a standard feedfoward network with *T* layers
  - But where the weights of each unfolded layer are the same (shared)
- We then train the unfolded *T* layer feedforward net with standard BP
- How to choose T?
  - Cross Validation, just like finding best number of hidden nodes, etc., thus we can find a good *T* fairly reasonably for a given task

Backpropagation through time (BPTT)

- Forward propagation over time computes hidden state vector and output at each time step.
- Backward propagation over time computes gradients for the weights at each time step
- Update the weights with the aggregated gradients computed at different times for each weight.

Forward propagation



For t=1 to T

$\mathbf{H}_t = \tanh(\mathbf{W}^X \mathbf{X}_t + \mathbf{W}^H \mathbf{H}_{t-1} + \mathbf{W}^H{}_0)$

$\hat{\mathbf{Y}}_t = \text{softmax}(\mathbf{W}^Y \mathbf{H}_t + \mathbf{w}^Y{}_0)$

Initialize H_0=0.5 or learn it

Forward propagation

Loss function:

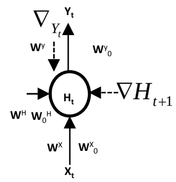
- The total loss for a given input/target sequence pair (**X, Y**) can be measured as the sum of loss at each time t, i.e.,

$$L(\mathbf{X},\mathbf{Y}) = \sum_{t=1}^{T} l(\widehat{\mathbf{Y}}_t, \mathbf{Y}_t)$$

where *l*() can be the squared loss or cross-entroy loss function

Back propagation over time

- Compute gradients
  - Output gradient at each time



$$\nabla_{Y_t} = \frac{\partial l(\hat{Y}_t, Y_t)}{\partial Y_t}, \ \nabla_{H_t} = \frac{\partial \hat{Y}_t}{\partial H_t}\nabla_{Y_t} + \frac{\partial H_{t+1}}{\partial H_t}\nabla_{H_{t+1}}$$

  - Weight gradients at each time

$$\nabla_{W^Y} = \frac{\partial \hat{Y}_t}{\partial W^Y}\nabla Y_t, \ \nabla_{W_0^Y} = \frac{\partial \hat{Y}_t}{\partial W_0^Y}\nabla Y_t, \ \nabla_{W^H} = \frac{\partial H^t}{\partial W^H}\nabla H_t$$

$$\nabla_{W_0^H} = \frac{\partial H^t}{\partial W_0^H}\nabla H_t, \ \nabla_{W^X} = \frac{\partial H^t}{\partial W^X}\nabla H_t, \ \nabla_{W_0^X} = \frac{\partial H^t}{\partial W_0^X}\nabla H_t$$

Back propagation over time

- Compute weight gradients over all times

$$\nabla_{W^Y} = \sum_{t=1}^{T}\frac{\partial \hat{Y}_t}{\partial W^Y}\nabla Y_t, \ \nabla_{W_0^Y} = \sum_{t=1}^{T}\frac{\partial \hat{Y}_t}{\partial W_0^Y}\nabla Y_t, \ \nabla_{W^H} = \sum_{t=1}^{T}\frac{\partial H_t}{\partial W^H}\nabla H_t$$

$$\nabla_{W_0^H} = \sum_{t=1}^{T}\frac{\partial H_t}{\partial W_0^H}\nabla H_t, \nabla_{W^X} = \sum_{t=1}^{T}\frac{\partial H_t}{\partial W^X}\nabla H_t, \nabla_{W_0^X} = \sum_{t=1}^{T}\frac{\partial H_t}{\partial W_0^X}\nabla H_t$$

Updating Weights

$$W^Y(k) = W^Y(k-1) - \eta_y \nabla_{W^Y}$$

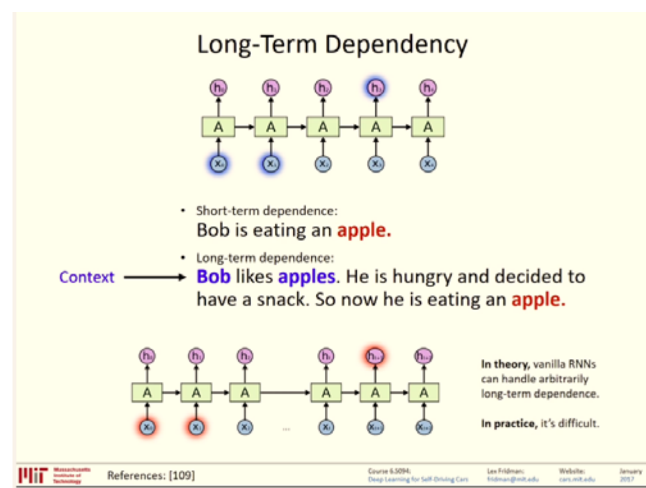
$$W_0^Y(k) = W_0^Y(k-1) - \eta_{y0} \nabla_{W_0^Y}$$

$$W^H(k) = W^H(k-1) - \eta_h \nabla_{W^H}$$

$$W_0^H(k) = W_0^H(k-1) - \eta_{h0} \nabla_{W_0^H}$$

$$W^X(k) = W^X(k-1) - \eta_x \nabla_{W^X}$$

$$W_0^X(k) = W_0^X(k-1) - \eta_{x0} \nabla_{W_0^X}$$

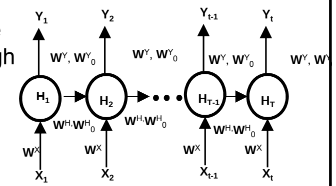


Exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?

At t = 1

$$\nabla_{W^H} = \frac{\partial H^1}{\partial W^H}\underbrace{\left(\frac{\partial H^2}{\partial H^1}\frac{\partial H^3}{\partial H^2}\cdots\frac{\partial H^T}{\partial H^{T-1}}\right)}$$

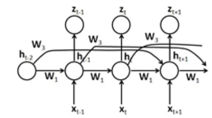


- In an RNN trained on long sequences (*e.g.* *T*=100 time steps), the gradient could easily explode when Jacobian matrix ($\partial H^{t+1}/\partial H^t$) is much larger than 1 or vanish when it is much less than 1
  - So RNNs learning via BPTT has difficulty dealing with long-range dependencies-which means it cannot remember inputs in the long past though RNN can encode all past inputs

Slide from Geoffrey Hinton

Effective ways to learn an RNN

- Hessian Free Optimization: Deal with the vanishing gradients problem by using a fancy optimizer that can detect directions with a tiny gradient but even smaller curvature.
  - The HF optimizer (Martens & Sutskever, 2011) is good at this.
- Good initialization with momentum
  Initialize very carefully and learn all of the connections using momentum.
- Gradient clipping: deal with gradient exploding. If gradient is much larger than a threshold, set it equal to the L2-normal
- Higher order (skip) connections
  Direct connections of the long past states to current state



- Long Short Term Memory
  Incorporate the RNN with a special memory cell that can remember values for a long time

Slide from Geoffrey Hinton