Introduction to Deep Learning

Course Instructor: Dr. M. Umair
Topic: Recurrent Neural Networks

Agenda

1. Preliminaries
2. Recurrent neural networks (RNN)
3. How RNNs Work
4. Computational Graph of RNN
5. RNN Designs
6. Backpropagation Through Time
7. Teacher Forcing Training Technique
8. Problems of RNNs
9. Code Example

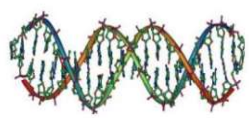
Preliminaries

Preliminaries

- **What Problems are CNNs normally good at?**
  - Image classification as a naive example.
    - Input: one image
    - Output: the probability distribution of classes
  - CNN provides *one guess (output)*, and to do that it only need to look at *one image (input)*.

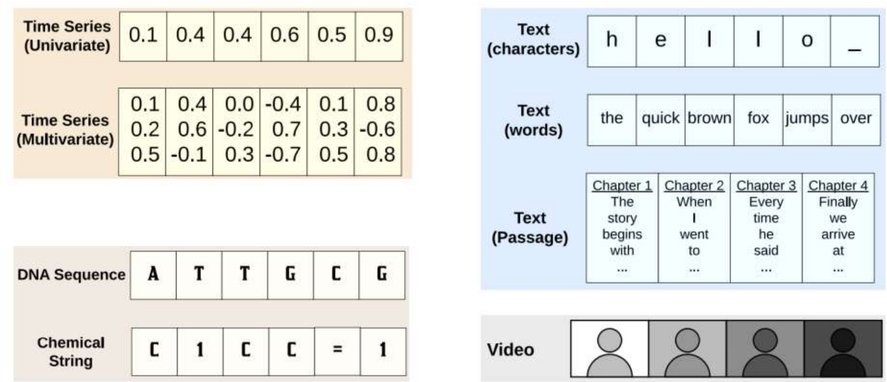
Preliminaries

- **What About Sequential Learning?**
  - SEQUENCE LEARNING is the study of machine learning algorithms designed for *sequential data.*
  - Sequential Data
    - Each data point is a sequence of vectors $x(t)$, *for* $1 \leq t \leq \tau$
    - *Batch data* is many sequences with different lengths τ



Preliminaries

- **Examples of Sequential Data**



Recurrent Neural Networks

Recurrent Neural Networks

- **Introduction**
  - RECURRENT NEURAL NETWORKS or RNNs are a family of neural networks for *processing sequential data* which involves *variable length* inputs or outputs.
  - Dates back to (Rumelhart et al., 1986).
  - RECALL: A *CNN* is a neural network that is specialized for *processing a grid of values, such as an image.*
  - *RNN* is a neural network that is specialized for *processing a sequence of values* $x(i)$, ..., $x(\tau)$. Especially, for natural language processing (NLP).

Recurrent Neural Networks

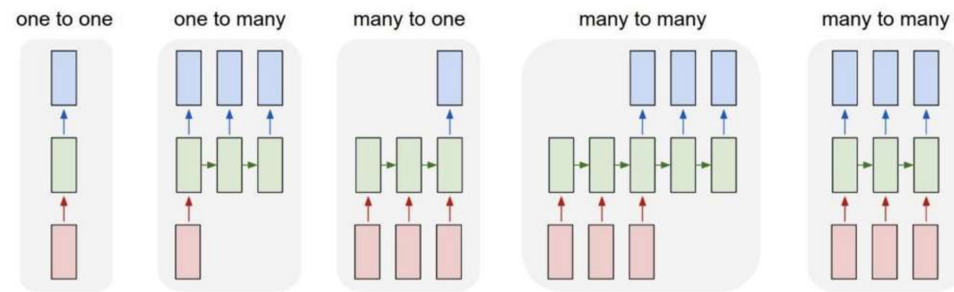
- **CNN Vs RNN**
  - CNN is *convolution in space.*
  - RNN is *convolution in time.*

CNN models relationships in space. Filter slides along *x and y dimensions.*

RNN models relationships in time. "Filter" slides along *time dimension.*

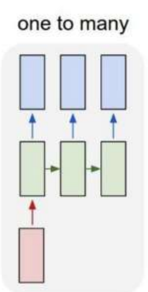
Recurrent Neural Networks

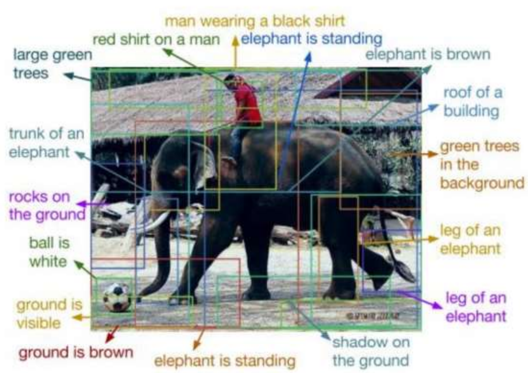
- **Types of RNNs**
  - Different types of RNNs are presented. *Red boxes* are input vectors. *Green boxes* are hidden layers. *Blue boxes* are output vectors.



Recurrent Neural Networks

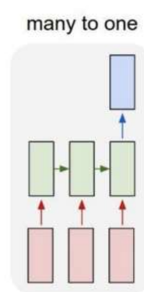
- **One to Many - Example**

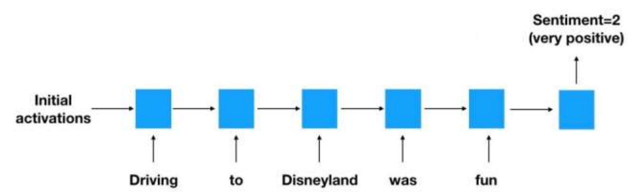




Recurrent Neural Networks

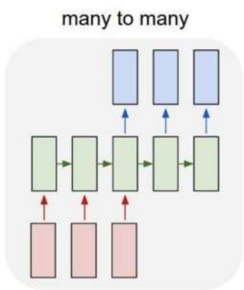
- **Many to One - Example**

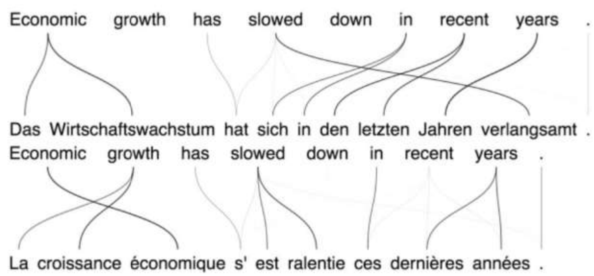




Recurrent Neural Networks

- **Many to Many - Example**

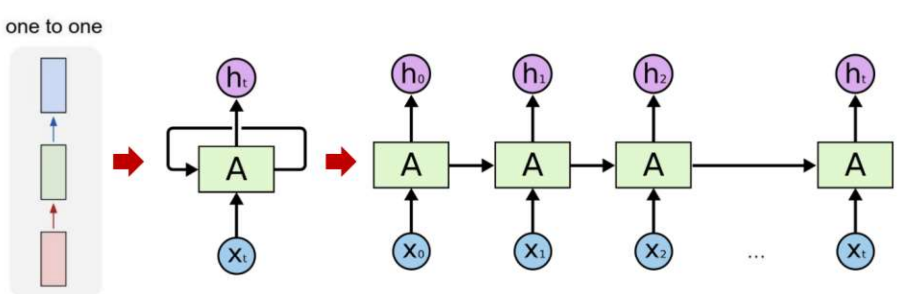




How RNNs Work

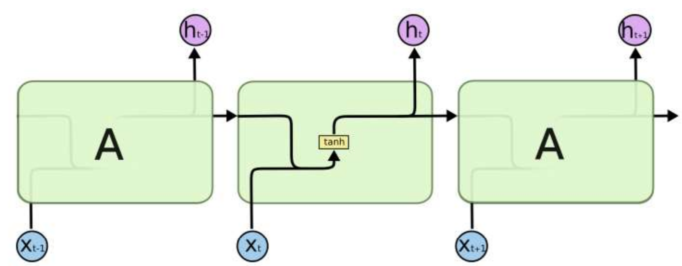
How RNNs Work

- **One-to-One (Vanilla) RNN as an Example**
  - Unrolling one-to-one RNN – *Simplified Version*



How RNNs Work

- **One-to-One (Vanilla) RNN as an Example**
  - In standard RNNs, this module will have a very simple structure, such as a single *tanh* layer. However, *sigmoid* is also a choice.



How RNNs Work

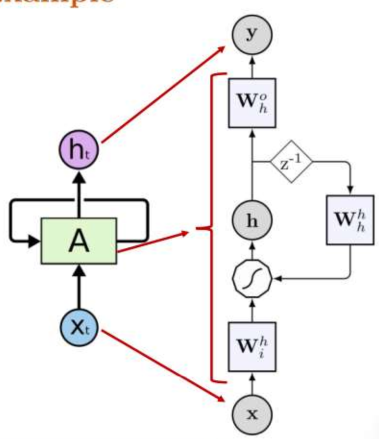
- **Why Not ReLU ?**
  - *RNNs* process *sequences of data* and *maintain hidden states* that are updated at each time step.
  - *If the hidden state values become very large* (which can happen due to the nature of ReLU allowing positive values to grow unbounded), the *gradients* during backpropagation can also become *excessively large.*
  - This phenomenon leads to *exploding gradients* and potentially *resulting in NaN values.*

How RNNs Work

- **Why Not ReLU ?**
  - ReLU outputs *zero* for any *negative input*. In an RNN context, if a *neuron consistently receives negative inputs, it will output zero*, effectively becoming inactive or "dying." This can lead to *a situation where entire neurons do not contribute to learning.*
  - The *output of ReLU is unbounded* (i.e., $[0,\infty]$) which can lead to *issues with maintaining stable hidden states* across time steps. In contrast, functions like *tanh have a bounded output range* of $[-1,1]$ which helps in stabilizing the learning process by keeping activations within a manageable range.

How RNNs Work

- **One-to-One (Vanilla) RNN as an Example**
  - Architecture Components
    - x: input
    - y: output
    - h: internal state (memory of the network)
    - W_i^h: input weights
    - W_h^h: recurrent layer weights
    - W_h^o: output weights
    - z^{-1}: time-delay unit
    - : neuron transfer function



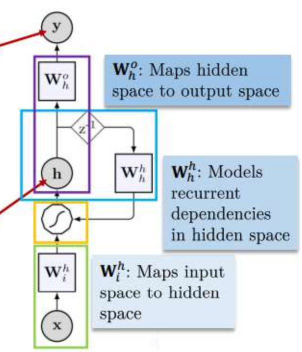
How RNNs Work

- **One-to-One (Vanilla) RNN as an Example**
  - Equations of RNN state and output are as follow:

$$y[t+1] = g\left(W_h^o h[t+1] + b_o\right)$$

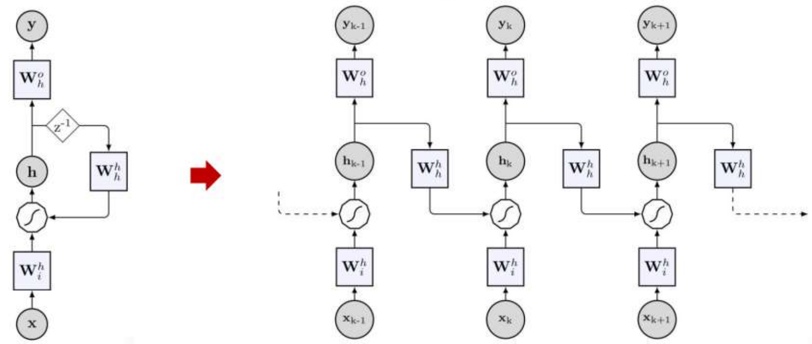
g() is either identity function or softmax function.

$$h[t+1] = f\left(W_h^h h[t] + W_i^h x[t+1] + b_h\right)$$



How RNNs Work

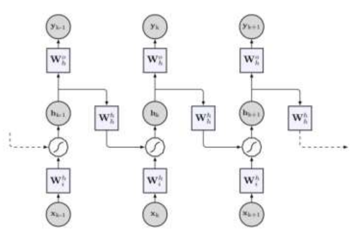
- **One-to-One (Vanilla) RNN as an Example**
  - Unrolling one-to-one RNN – *Detailed Version*



The idea is to "**share parameters across different parts of a model.**"

How RNNs Work

- **One-to-One (Vanilla) RNN as an Example**
  - Advantages of RNN
    - Overcomes problem that "weights of each layer are learned independently" *by using previous hidden state.*
    - Model has *less parameters* since weights are shared across layers.
    - *Retains information* about past inputs for an amount of *time* that *depends* on the model's *weights* and *input* data rather than a fixed duration selected a priori.



Computational Graph of RNN

Computational Graph of RNN

- **Computational Graph**
  - A COMPUTATIONAL GRAPH is a way to *formalize the structure of a set of computations*, such as those involved in *mapping inputs and parameters to outputs and loss.*
  - Consider the classical form of a *dynamical system* where $s^{(t)}$ is called the *state of the system*.

$$\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)}; \boldsymbol{\theta})$$

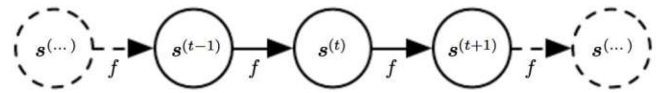
$S(t)$ = hidden state at time step t
θ= network parameter

  - The *equation is recurrent* because the definition of s at time t refers back to the same definition at time t-1.

Computational Graph of RNN

- **Computational Graph**
  - If we unfold the equation for $t = 3$ time steps, we obtain

$$\begin{aligned}\boldsymbol{s}^{(3)} &= f(\boldsymbol{s}^{(2)}; \boldsymbol{\theta}) \\ &= f(f(\boldsymbol{s}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta})\end{aligned}$$



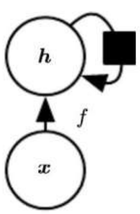
An unfolded graph. Same value of θ is used to parametrize the function f.

Computational Graph of RNN

- **RNN General Equation & Computational Graph**

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

$h^{(t)}$ = hidden state at time step t
$x^{(t)}$ = external signal
Θ = network parameter

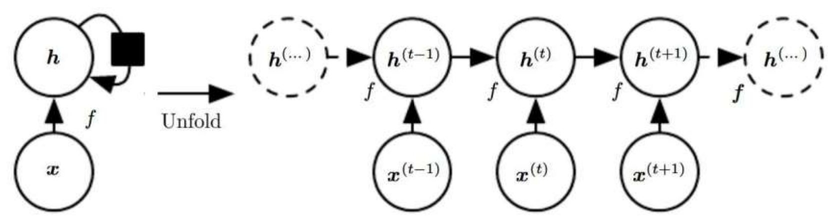


Indicate that an interaction takes place with a delay of a single time step.

Computational Graph of RNN

- **RNN General Equation & Computational Graph**

$$\boldsymbol{h}^{(t)} = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$$

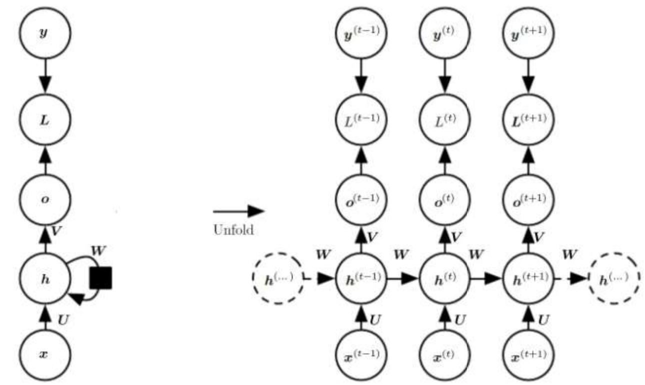


A recurrent network with no outputs

RNN Designs

RNN Designs

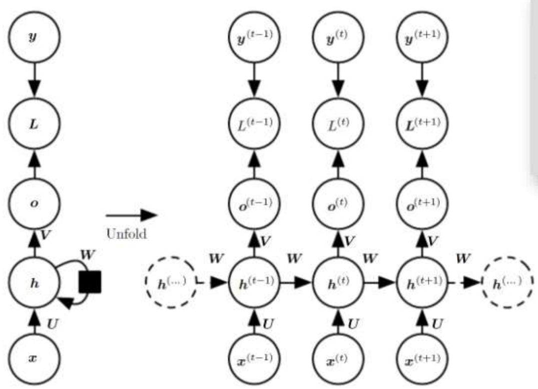
- **RNN Design 1**



$x =$ input sequence
$h =$ hidden state
$o =$ output
$L =$ loss
$y =$ training target
$\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W} =$ weight matrices

RNN Designs

- **RNN Design 1 - Equations**



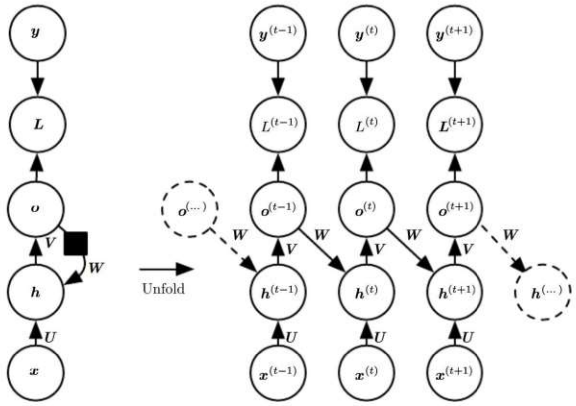
$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

$\boldsymbol{b},\ \boldsymbol{c} =$ bias
$\boldsymbol{W},\ \boldsymbol{U},\ \boldsymbol{V} =$ weight matrices

This design is **more powerful** as it can choose to **put any information** it wants about the past into its **hidden representation** h and **transmit** h to the **future**.

RNN Designs

- **RNN Design 2**



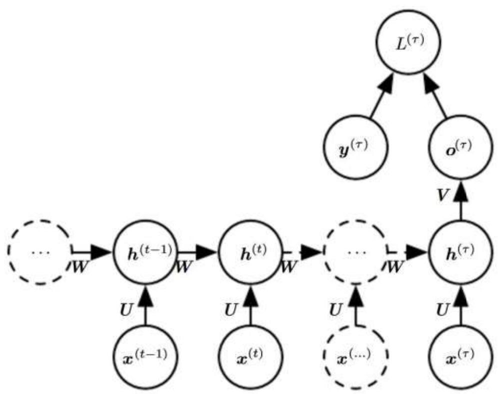
x = input sequence
h = hidden state
o = output
L = loss
y = training target
$\boldsymbol{U, V, W}$ = weight matrices

This design is **less powerful** as it put
a **specific output value** into o,
and o **is the only information** it is allowed to send to the future.
There are **no direct connections from** h going forward.

Unless o is very high-dimensional and rich

RNN Designs

- **RNN Design 3**



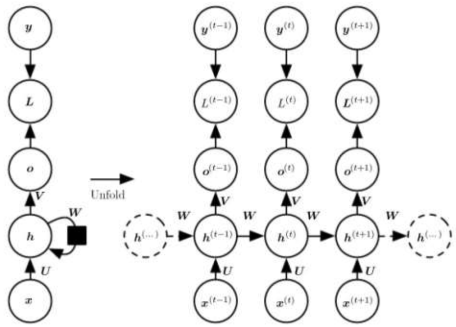
x = input sequence
h = hidden state
o = output
L = loss
y = training target
$\boldsymbol{U, V, W}$ = weight matrices

Such a network can be used to summarize a sequence.

Back Propagation Through Time

Back Propagation Through Time

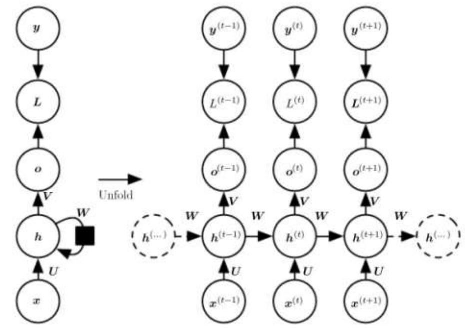
- **Introduction**
  - Back Propagation Through Time (BPTT) is the *adaption of the backpropagation* algorithm for RNNs.
  - In theory, this *unfolds the RNN to construct a traditional feedforward neural network* where we can apply back propagation.
  - BPTT uses the same idea as regular backpropagation *(adjust weights using gradients)* but it also *considers how errors from future steps depend on past steps.*



Back Propagation Through Time

- **How it Works?**
  - When we forward pass our *input* $\boldsymbol{X}_t$ through the network we *compute the hidden state* $\boldsymbol{H}_t$ and the *output state* $\boldsymbol{O}_t$ *one step at a time.*
  - We can then define a *loss function* $\mathcal{L}(\boldsymbol{O},\boldsymbol{Y})$ to describe the difference between all *outputs* $\boldsymbol{O}_t$ and *target values* $\boldsymbol{Y}_t$ as shown in equation.

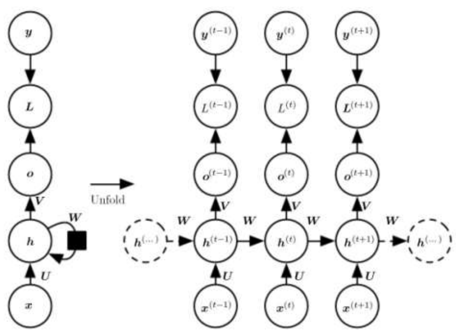
$$\mathcal{L}(\mathbf{O},\mathbf{Y}) = \sum_{t=1}^{T} \ell_t(\mathbf{O}_t,\mathbf{Y}_t)$$



Back Propagation Through Time

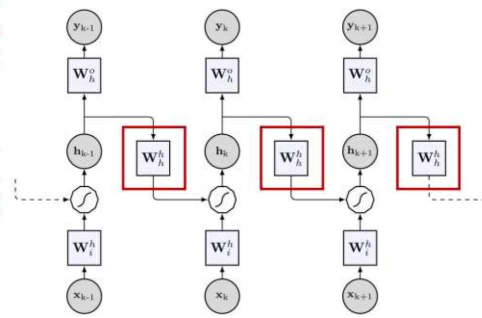
- **How it Works?**
  - This basically *sums up every loss term* ℓ_t of each update step so far.
  - This *loss term* ℓ_t can have *different definitions based on the specific problem* (e.g. Mean Squared Error, Hinge Loss, Cross Entropy Loss, etc.).

$$\mathcal{L}(\mathbf{O},\mathbf{Y}) = \sum_{t=1}^{T} \ell_t(\mathbf{O}_t,\mathbf{Y}_t)$$



Back Propagation Through Time

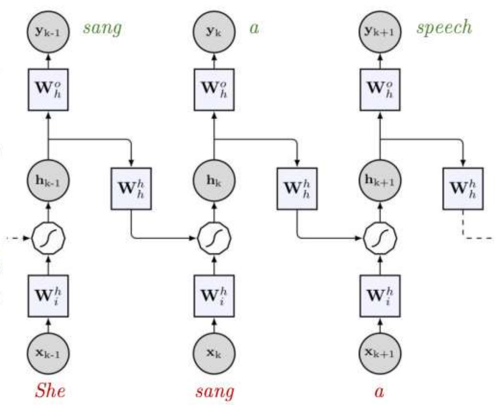
- **How it Works?**
  - Since we have *three weight matrices* $\boldsymbol{W}_i^h$, $\boldsymbol{W}_h^h$, *and* $\boldsymbol{W}_h^o$ we need to *compute the partial derivative* w.r.t. to each of these weight matrices.
  - $\boldsymbol{W}_h^h$ the recurrent weight matrix, is particularly *notable* because it is *responsible for learning the temporal dependencies in the sequence.*



Back Propagation Through Time

- **How it Works?**

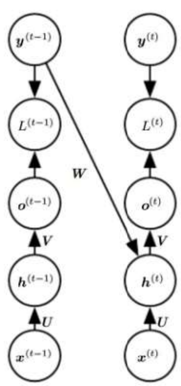
1. *Start* at the *mistake* (speech). *Calculate* how far "speech" is from the *correct* word "song".
2. *Adjust the weights* at the last step (where *"a" led to the mistake*).
3. *Backtrack* to the earlier steps ("sang", "She") to *adjust weights so that the context is better captured* in future predictions.



Teacher Forcing Training Technique

Teacher Forcing Training Technique

- **Computational Graph**



x = input sequence
h = hidden state
o = output
L = loss
y = training target
$\boldsymbol{U}, \boldsymbol{V}, \boldsymbol{W}$ = weight matrices

Problems of RNNs

Problems of RNNs

- **Vanishing & Exploding Gradients**
  - As in most neural networks, *vanishing* or *exploding gradients* is a key problem of RNNs.
  - Back propagation through time introduces *matrix multiplication* over the (potentially very long) sequence.

Problems of RNNs

- **Vanishing & Exploding Gradients**
  - *If there are small values (< 1) in the matrix* multiplication this causes the gradient to *decrease* with each layer (or time step) and *finally vanish.*
  - This basically *stops the contribution of states that happened far earlier* than the current time step towards the current time step.

Problems of RNNs

- **Vanishing & Exploding Gradients**
  - This can happen in the opposite direction *if we have large values (> 1)* during matrix multiplication
  - This leads us to an *exploding gradient* which in result values each weight too much and *changes it heavily.*

Problems of RNNs

- **Solutions: Truncated Backpropagation Through Time**
  - Instead of backpropagating through the entire sequence, Truncated Backpropagation Through Time (TBPTT) *limits* the backpropagation to a *fixed number of time steps.*
    - This *reduces memory usage* by not storing all hidden states for long sequences, allowing for manageable computation without losing significant context.
  - Memory-Efficient architectures such as *long short-term memory* (LSTMs) are designed to handle longer sequences more effectively by mitigating issues like vanishing gradients.

Code Example

Code Example

- **Recurrent Neural Networks**
  - https://www.tensorflow.org/guide/keras/working_with_rnns

References

References

1. https://cs231n.github.io/rnn/
2. Recurrent Neural Network, TINGWU WANG
3. RECURRENT NEURAL NETWORKS, A Quick Review, Filippo Maria Bianchi
4. Recurrent Neural Networks, Vedant Sanil, David Park
5. Sequential Interpretability: Methods, Applications, and Future Direction for Understanding Deep Learning Models in the Context of Sequential Data, Benjamin Shickel et al.
6. Deep Learning, Ian Goodfellow et al.
7. Recurrent Neural Networks (RNNs): A gentle Introduction and Overview, Robin M. Schmidt, 2019