Introduction

Sequence Learning with Neural Networks
Some Sequence Tasks

Figure credit: Andrej Karpathy
Problems with MLPs for Sequence Tasks

- The "API" is too limited.
Problems with MLPs for Sequence Tasks

- The "API" is too limited.
- MLPs only accept an input of fixed dimensionality and map it to an output of fixed dimensionality.
Problems with MLPs for Sequence Tasks

- The "API" is too limited.
- MLPs only accept an input of fixed dimensionality and map it to an output of fixed dimensionality.
- Great e.g.: Inputs - Images, Output - Categories.
Problems with MLPs for Sequence Tasks

- The "API" is too limited.
- MLPs only accept an input of fixed dimensionality and map it to an output of fixed dimensionality.
- Great e.g.: Inputs - Images, Output - Categories.
- Bad e.g.: Inputs - Text in one language, Output - Text in another language.
Problems with MLPs for Sequence Tasks

- The "API" is too limited.
- MLPs only accept an input of fixed dimensionality and map it to an output of fixed dimensionality
- Great e.g.: Inputs - Images, Output - Categories
- Bad e.g.: Inputs - Text in one language, Output - Text in another language
- MLPs treat every example independently. How is this problematic?
Problems with MLPs for Sequence Tasks

- The "API" is too limited.
- MLPs only accept an input of fixed dimensionality and map it to an output of fixed dimensionality.
- Great e.g.: Inputs - Images, Output - Categories
- Bad e.g.: Inputs - Text in one language, Output - Text in another language
- MLPs treat every example independently. How is this problematic?
- Need to re-learn the rules of language from scratch each time.
Problems with MLPs for Sequence Tasks

- The "API" is too limited.
- MLPs only accept an input of fixed dimensionality and map it to an output of fixed dimensionality.
- Great e.g.: Inputs - Images, Output - Categories
- Bad e.g.: Inputs - Text in one language, Output - Text in another language
- MLPs treat every example independently. How is this problematic?
- Need to re-learn the rules of language from scratch each time
- Another example: Classify events after a fixed number of frames in a movie
Problems with MLPs for Sequence Tasks

- The "API" is too limited.
- MLPs only accept an input of fixed dimensionality and map it to an output of fixed dimensionality
- Great e.g.: Inputs - Images, Output - Categories
- Bad e.g.: Inputs - Text in one language, Output - Text in another language
- MLPs treat every example independently. How is this problematic?
- Need to re-learn the rules of language from scratch each time
- Another example: Classify events after a fixed number of frames in a movie
- Need to re-use knowledge about the previous events to help in classifying the current.
Recurrent Networks

- Recurrent Neural Networks (Rumelhart, 1986) are a family of neural networks for handling sequential data.
Recurrent Networks

- Recurrent Neural Networks (Rumelhart, 1986) are a family of neural networks for handling sequential data.
- Sequential data: Each example consists of a pair of sequences. Each example can have different lengths.

Need to take advantage of an old idea in Machine Learning:
- Share parameters across different parts of a model.
- Makes it possible to extend the model to apply it to sequences of different lengths not seen during training.

Without parameter sharing it would not be possible to share statistical strength and generalize to lengths of sequences not seen during training.
Recurrent Neural Networks (Rumelhart, 1986) are a family of neural networks for handling sequential data.

Sequential data: Each example consists of a pair of sequences. Each example can have different lengths.

Need to take advantage of an old idea in Machine Learning: Share parameters across different parts of a model.
Recurrent Networks

- Recurrent Neural Networks (Rumelhart, 1986) are a family of neural networks for handling sequential data.
- Sequential data: Each example consists of a pair of sequences. Each example can have different lengths.
- Need to take advantage of an old idea in Machine Learning: Share parameters across different parts of a model.
- Makes it possible to extend the model to apply it to sequences of different lengths not seen during training.
Recurrent Neural Networks (Rumelhart, 1986) are a family of neural networks for handling sequential data.

Sequential data: Each example consists of a pair of sequences. Each example can have different lengths.

Need to take advantage of an old idea in Machine Learning: Share parameters across different parts of a model.

Makes it possible to extend the model to apply it to sequences of different lengths not seen during training.

Without parameter sharing it would not be possible to share statistical strength and generalize to lengths of sequences not seen during training.
Recurrent Networks

- Recurrent Neural Networks (Rumelhart, 1986) are a family of neural networks for handling sequential data.
- Sequential data: Each example consists of a pair of sequences. Each example can have different lengths.
- Need to take advantage of an old idea in Machine Learning: Share parameters across different parts of a model.
- Makes it possible to extend the model to apply it to sequences of different lengths not seen during training.
- Without parameter sharing it would not be possible to share statistical strength and generalize to lengths of sequences not seen during training.
- Recurrent networks share parameters: Each output is a function of the previous outputs, with the same update rule applied.
Recurrence

Consider the classical form of a dynamical system:

\[ s(t) = f(s(t-1); \theta) \]
Recurrence

- Consider the classical form of a dynamical system:

\[ s(t) = f(s(t-1); \theta) \]

- This is recurrent because the definition of \( s \) at time \( t \) refers back to the same definition at time \( t - 1 \)
Recurrence

- Consider the classical form of a dynamical system:

\[ s(t) = f(s(t-1); \theta) \]

- This is recurrent because the definition of \( s \) at time \( t \) refers back to the same definition at time \( t - 1 \)

- For some finite number of time steps \( \tau \), the graph represented by this recurrence can be unfolded by using the definition \( \tau - 1 \) times. For example when \( \tau = 3 \)
Recurrence

- Consider the classical form of a dynamical system:
  \[ s^{(t)} = f(s^{(t-1)}; \theta) \]

- This is recurrent because the definition of \( s \) at time \( t \) refers back to the same definition at time \( t - 1 \)

- For some finite number of time steps \( \tau \), the graph represented by this recurrence can be unfolded by using the definition \( \tau - 1 \) times. For example when \( \tau = 3 \)
  \[ s^{(3)} = f(s^{(2)}; \theta) = f(f(s^{(1)}; \theta); \theta) \]
Recurrence

Consider the classical form of a dynamical system:

\[ s^{(t)} = f(s^{(t-1)}; \theta) \]

This is recurrent because the definition of \( s \) at time \( t \) refers back to the same definition at time \( t - 1 \).

For some finite number of time steps \( \tau \), the graph represented by this recurrence can be unfolded by using the definition \( \tau - 1 \) times. For example when \( \tau = 3 \)

\[ s^{(3)} = f(s^{(2)}; \theta) = f(f(s^{(1)}; \theta); \theta) \]

This expression does not involve any recurrence and can be represented by a traditional directed acyclic computational graph.
Recurrent Networks

Consider another dynamical system, that is driven by an external signal $x(t) = f(s(t-1), x(t); \theta)$.

The state now contains information about the whole past sequence.

RNNs can be built in various ways: Just as any function can be considered a feedforward network, any function involving a recurrence can be considered a recurrent neural network.
Consider another dynamical system, that is driven by an external signal $x(t)$.
Consider another dynamical system, that is driven by an external signal $x(t)$

$$s(t) = f(s(t-1), x(t); \theta)$$
Consider another dynamical system, that is driven by an external signal $x(t)$

$$s(t) = f(s(t-1), x(t); \theta)$$

The state now contains information about the whole past sequence.
Consider another dynamical system, that is driven by an external signal $x^{(t)}$

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

The state now contains information about the whole past sequence

RNNs can be built in various ways: Just as any function can be considered a feedforward network, any function involving a recurrence can be considered a recurrent neural network.
We can consider the states to be the hidden units of the network, so we replace $s(t)$ by $h(t)$.
We can consider the states to be the hidden units of the network, so we replace $s^{(t)}$ by $h^{(t)}$

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$
We can consider the states to be the hidden units of the network, so we replace $s(t)$ by $h(t)$

$$h(t) = f(h(t-1), x(t); \theta)$$

This system can be drawn in two ways:
We can consider the states to be the hidden units of the network, so we replace $s(t)$ by $h(t)$

$$h(t) = f(h(t-1), x(t); \theta)$$

This system can be drawn in two ways:

We can have additional architectural features: Such as output layers that read information from $h$ to make predictions
When the task is to predict the future from the past, the network learns to use $h(t)$ as a summary of task relevant aspects of the past sequence upto time $t$. This summary is lossy because it maps an arbitrary length sequence $(x(1), x(t-1), ..., x(2), x(1))$ to a fixed vector $h(t)$. Depending on the training criterion, the summary might selectively keep some aspects of the past sequence with more precision (e.g. statistical language modeling). Most demanding situation for $h(t)$: Approximately recover the input sequence.
When the task is to predict the future from the past, the network learns to use $h^{(t)}$ as a summary of task relevant aspects of the past sequence upto time $t$.

This summary is lossy because it maps an arbitrary length sequence $(x^{(1)}, x^{(t-1)}, \ldots, x^{(2)}, x^{(1)})$ to a fixed vector $h^{(t)}$. 
When the task is to predict the future from the past, the network learns to use $h(t)$ as a summary of task relevant aspects of the past sequence up to time $t$.

This summary is lossy because it maps an arbitrary length sequence $(x^{(1)}, x^{(t-1)}, \ldots, x^{(2)}, x^{(1)})$ to a fixed vector $h(t)$.

Depending on the training criterion, the summary might selectively keep some aspects of the past sequence with more precision (e.g., statistical language modeling).
When the task is to predict the future from the past, the network learns to use $h^{(t)}$ as a summary of task relevant aspects of the past sequence up to time $t$.

This summary is lossy because it maps an arbitrary length sequence $(x^{(1)}, x^{(t-1)}, \ldots, x^{(2)}, x^{(1)})$ to a fixed vector $h^{(t)}$.

Depending on the training criterion, the summary might selectively keep some aspects of the past sequence with more precision (e.g. statistical language modeling).

Most demanding situation for $h^{(t)}$: Approximately recover the input sequence.
Plain Vanilla RNN: Produce an output at each time stamp and have recurrent connections between hidden units
Plain Vanilla RNN: Produce an output at each time stamp and have recurrent connections between hidden units. Is infact Turing Complete (Siegelmann, 1991, 1995, 1995)
Design Patterns of Recurrent Networks
Plain Vanilla Recurrent Network

\[ y_t \]

\[ h_t \]

\[ x_t \]
Recurrent Connections

\[ Y_t = \psi(U_{xt} + W_{h_{t-1}}) \]

Diagram: Diagram showing the flow of information in a recurrent neural network, with inputs and hidden states connected by recurrent connections.
Recurrent Connections

\[
\hat{y}_t = \phi(V h_t)
\]

\[
h_t = \psi(U x_t + W h_{t-1})
\]

\[\psi\] can be \text{tanh} and \[\phi\] can be \text{softmax}
Unrolling the Recurrence

\[ x_1, x_2, x_3, \ldots, x_\tau \]
Unrolling the Recurrence

\[ U \]

\[ x_1 \ x_2 \ x_3 \ \cdots \ x_T \]
Unrolling the Recurrence
Unrolling the Recurrence

\[
\hat{y}_1
\]

\[
V
\]

\[
\begin{array}{c}
\text{h}_1 \\
U
\end{array}
\]

\[
W
\]

\[
\begin{array}{c}
h_2 \\
U
\end{array}
\]

\[
\begin{array}{c}
x_1
\end{array}
\]

\[
\begin{array}{c}
x_2
\end{array}
\]

\[
\begin{array}{c}
x_3
\end{array}
\]

\[
\cdots
\]

\[
\begin{array}{c}
x_T
\end{array}
\]
Unrolling the Recurrence

\[
\begin{align*}
\hat{y}_1 & \quad V \\
& \quad W \\
& \quad U \\
\hat{y}_2 & \quad V \\
& \quad W \\
& \quad U \\
x_1 & \quad \ldots \\
x_2 & \quad \ldots \\
x_3 & \quad \ldots \\
x_{\tau} & \quad \ldots 
\end{align*}
\]
Unrolling the Recurrence

Unrolling the Recurrence means unfolding the recurrence relation into a sequence of steps. This process is often used in the context of Recurrent Neural Networks (RNNs) to handle sequences of data. The figure shows an RNN with unrolled layers, where each layer is connected by weights $W$ and bias terms. The input $x_t$ at each time step $t$ is transformed into the hidden state $h_t$ and then to the output $y_t$. The process is repeated for each time step up to $T$.
Unrolling the Recurrence

\[ \hat{y}_1 \quad \hat{y}_2 \quad \hat{y}_3 \]

\[ V \quad V \quad V \]

\[ h_1 \quad h_2 \quad h_3 \]

\[ W \quad W \]

\[ U \quad U \quad U \]

\[ x_1 \quad x_2 \quad x_3 \quad \ldots \quad x_T \]
Unrolling the Recurrence

\[ \hat{y}_1, \hat{y}_2, \hat{y}_3, \ldots, \hat{y}_T \]

\[ U, V, W \]

Lecture 11 Recurrent Neural Networks I CMSC 35246
Unrolling the Recurrence

\[ y_1 \xrightarrow{U} h_1 \xrightarrow{W} h_2 \xrightarrow{W} h_3 \ldots \xrightarrow{W} h_\tau \]

\[ V \]

\[ x_1 \xrightarrow{U} h_1 \xrightarrow{W} h_2 \xrightarrow{W} h_3 \ldots \xrightarrow{W} h_\tau \]
Unrolling the Recurrence

\[ \hat{y}_1, \hat{y}_2, \hat{y}_3, \ldots, \hat{y}_\tau \]

\[ V \]

\[ h_1, h_2, h_3, \ldots, h_\tau \]

\[ W \]

\[ U \]

\[ x_1, x_2, x_3, \ldots, x_\tau \]

Lecture 11 Recurrent Neural Networks I
Feedforward Propagation

- This is a RNN where the input and output sequences are of the same length
Feedforward Propagation

- This is a RNN where the input and output sequences are of the same length
- Feedforward operation proceeds from left to right

Update Equations:

\[ a_t = b + W_h t_{t-1} + U_x t \]
\[ h_t = \tanh(a_t) \]
\[ o_t = c + V_h t \]
\[ \hat{y}_t = \text{softmax}(o_t) \]
Feedforward Propagation

- This is a RNN where the input and output sequences are of the same length
- Feedforward operation proceeds from left to right
- Update Equations:
Feedforward Propagation

- This is a RNN where the input and output sequences are of the same length
- Feedforward operation proceeds from left to right
- Update Equations:

\[ a_t = b + Wh_{t-1} + Ux_t \]
\[ h_t = \tanh a_t \]
\[ o_t = c + Vh_t \]
\[ \hat{y}_t = \text{softmax}(o_t) \]
Feedforward Propagation

- Loss would just be the sum of losses over time steps

\[ L_t = \text{the negative log-likelihood of } y_t \text{ given } x_1, \ldots, x_t, \text{ then:} \]

\[ L(\{x_1, \ldots, x_t\}, \{y_1, \ldots, y_t\}) = \sum_t L_t \]

With:

\[ \sum_t L_t = -\sum_t \log p_{\text{model}}(y_t | \{x_1, \ldots, x_t\}) \]

Observation: Forward propagation takes time \( O(t) \); can't be parallelized
Loss would just be the sum of losses over time steps

If $L_t$ is the negative log-likelihood of $y_t$ given $x_1, \ldots, x_t$, then:
Feedforward Propagation

- Loss would just be the sum of losses over time steps
- If $L_t$ is the negative log-likelihood of $y_t$ given $x_1, \ldots, x_t$, then:

$$L \left( \{x_1, \ldots, x_t\}, \{y_1, \ldots, y_t\} \right) = \sum_t L_t$$
Feedforward Propagation

- Loss would just be the sum of losses over time steps.
- If $L_t$ is the negative log-likelihood of $y_t$ given $x_1, \ldots, x_t$, then:

$$L\left(\{x_1, \ldots, x_t\}, \{y_1, \ldots, y_t\}\right) = \sum_t L_t$$

- With:

$$\sum_t L_t = -\sum_t \log p_{\text{model}}(y_t | \{x_1, \ldots, x_t\})$$
Feedforward Propagation

- Loss would just be the sum of losses over time steps
- If $L_t$ is the negative log-likelihood of $y_t$ given $x_1, \ldots, x_t$, then:

$$L\left(\{x_1, \ldots, x_t\}, \{y_1, \ldots, y_t\}\right) = \sum_t L_t$$

- With:

$$\sum_t L_t = -\sum_t \log p_{\text{model}}(y_t|\{x_1, \ldots, x_t\})$$

- Observation: Forward propagation takes time $O(t)$; can’t be parallelized
Backward Propagation

Need to find: $\nabla V L$, $\nabla W L$, $\nabla U L$
Backward Propagation

- Need to find: $\nabla_V L$, $\nabla_W L$, $\nabla_U L$
- And the gradients w.r.t biases: $\nabla_c L$ and $\nabla_b L$
Backward Propagation

- Need to find: $\nabla_V L$, $\nabla_W L$, $\nabla_U L$
- And the gradients w.r.t biases: $\nabla_c L$ and $\nabla_b L$
- Treat the recurrent network as a usual multilayer network and apply backpropagation on the unrolled network
Backward Propagation

- Need to find: $\nabla_V L$, $\nabla_W L$, $\nabla_U L$
- And the gradients w.r.t biases: $\nabla_c L$ and $\nabla_b L$
- Treat the recurrent network as a usual multilayer network and apply backpropagation on the unrolled network
- We move from the right to left: This is called Backpropagation through time
Backward Propagation

- Need to find: \( \nabla_V L, \nabla_W L, \nabla_U L \)
- And the gradients w.r.t biases: \( \nabla_c L \) and \( \nabla_b L \)
- Treat the recurrent network as a usual multilayer network and apply backpropagation on the unrolled network
- We move from the right to left: This is called Backpropagation through time
- Also takes time \( O(t) \)
BPTT

\[
\hat{y}_1 \xrightarrow{V} h_1 \xrightarrow{W} h_2 \xrightarrow{W} h_3 \xrightarrow{W} h_4 \xrightarrow{W} h_5 \xrightarrow{V} \hat{y}_5
\]

\[
\hat{y}_2 \xrightarrow{V} h_2 \xrightarrow{W} h_3 \xrightarrow{W} h_4 \xrightarrow{W} h_5 \xrightarrow{V} \hat{y}_5
\]

\[
\hat{y}_3 \xrightarrow{V} h_3 \xrightarrow{W} h_4 \xrightarrow{W} h_5 \xrightarrow{V} \hat{y}_5
\]

\[
\hat{y}_4 \xrightarrow{V} h_4 \xrightarrow{W} h_5 \xrightarrow{V} \hat{y}_5
\]
BPTT

\[ \hat{y}_1, \hat{y}_2, \hat{y}_3, \hat{y}_4, \hat{y}_5 \]

\[ V \]

\[ h_1, h_2, h_3, h_4, h_5 \]

\[ U \]

\[ x_1, x_2, x_3, x_4, x_5 \]

\[ \frac{\partial L}{\partial U}, \frac{\partial L}{\partial W}, \frac{\partial L}{\partial V} \]
BPTT
BPTT

\[
\begin{align*}
\hat{y}_1 & \quad \hat{y}_2 & \quad \hat{y}_3 & \quad \hat{y}_4 & \quad \hat{y}_5 \\
V & \quad V & \quad V & \quad V & \quad V \\
\begin{array}{c}
h_1 \\
X_1
\end{array} & \quad \begin{array}{c}
h_2 \\
X_2
\end{array} & \quad \begin{array}{c}
h_3 \\
X_3
\end{array} & \quad \begin{array}{c}
h_4 \\
X_4
\end{array} & \quad \begin{array}{c}
h_5 \\
X_5
\end{array} \\
W & \quad W & \quad W & \quad W & \quad W \\
\frac{\partial L}{\partial W} & \quad \frac{\partial L}{\partial W} & \quad \frac{\partial L}{\partial W} & \quad \frac{\partial L}{\partial W} & \quad \frac{\partial L}{\partial W} \\
U & \quad U & \quad U & \quad U & \quad U \\
\frac{\partial L}{\partial U} & \quad \frac{\partial L}{\partial U} & \quad \frac{\partial L}{\partial U} & \quad \frac{\partial L}{\partial U} & \quad \frac{\partial L}{\partial U}
\end{align*}
\]
Gradient Computation

\[ \nabla_V L = \sum_t (\nabla_{o_t} L) h_t^T \]
Gradient Computation

\[ \nabla_V L = \sum_t (\nabla_{o_t} L) h_t^T \]

Where:

\[ (\nabla_{o_t} L)_i = \frac{\partial L}{\partial o_t^{(i)}} = \frac{\partial L}{\partial L_t} \frac{\partial L_t}{\partial o_t^{(i)}} = \hat{y}_t^{(i)} - 1_{i,y_t} \]
BPTT

\[
\hat{y}_1, \hat{y}_2, \hat{y}_3, \hat{y}_4, \hat{y}_5
\]

\[
h_1 \xrightarrow{W} h_2 \xrightarrow{W} h_3 \xrightarrow{W} h_4 \xrightarrow{W} h_5
\]

\[
\frac{\partial L}{\partial U}, \frac{\partial L}{\partial V}, \frac{\partial L}{\partial W}
\]

\[
x_1, x_2, x_3, x_4, x_5
\]
Gradient Computation

\[ \nabla_W L = \sum_t \text{diag}(1 - (h_t)^2) (\nabla_{h_t} L) h^T_{t-1} \]

Where, for \( t = \tau \) (one descendant):

\[ (\nabla_{h_\tau} L) = V^T (\nabla_{o_\tau} L) \]

For some \( t < \tau \) (two descendants)

\[ (\nabla_{h_t} L) = \left( \frac{\partial h_{t+1}}{\partial h_t} \right)^T (\nabla_{h_{t+1}} L) + \left( \frac{\partial o_t}{\partial h_t} \right)^T (\nabla_{o_t} L) \]

\[ = W^T (\nabla_{h_{t+1}} L) \text{diag}(1 - h_{t+1}^2) + V (\nabla_{o_t} L) \]
BPTT
Gradient Computation

\[ \nabla_U L = \sum_t \text{diag}(1 - (h_t)^2) (\nabla_{h_t} L)x_t^T \]
Recurrent Neural Networks

- But weights are shared across different time stamps? How is this constraint enforced?
Recurrent Neural Networks

- But weights are shared across different time stamps? How is this constraint enforced?
- Train the network as if there were no constraints, obtain weights at different time stamps, average them
Design Patterns of Recurrent Networks

- **Summarization:** Produce a single output and have recurrent connections from output between hidden units.
**Summarization:** Produce a single output and have recurrent connections from output between hidden units.

Useful for summarizing a sequence (e.g. sentiment analysis)
We have considered RNNs in the context of a sequence of vectors $x(t)$ with $t = 1, \ldots, \tau$ as input.
Design Patterns: Fixed vector as input

- We have considered RNNs in the context of a sequence of vectors \( x^{(t)} \) with \( t = 1, \ldots, \tau \) as input.
- Sometimes we are interested in only taking a single, fixed sized vector \( x \) as input, that generates the \( y \) sequence.
We have considered RNNs in the context of a sequence of vectors $x(t)$ with $t = 1, \ldots, \tau$ as input.

Sometimes we are interested in only taking a single, fixed sized vector $x$ as input, that generates the $y$ sequence.

Some common ways to provide an extra input to an RNN are:
We have considered RNNs in the context of a sequence of vectors $x^{(t)}$ with $t = 1, \ldots, \tau$ as input.

Sometimes we are interested in only taking a single, fixed sized vector $x$ as input, that generates the $y$ sequence.

Some common ways to provide an extra input to an RNN are:

- As an extra input at each time step
We have considered RNNs in the context of a sequence of vectors $x^{(t)}$ with $t = 1, \ldots, \tau$ as input.

Sometimes we are interested in only taking a single, fixed sized vector $x$ as input, that generates the $y$ sequence.

Some common ways to provide an extra input to an RNN are:
- As an extra input at each time step
- As the initial state $h^{(0)}$
We have considered RNNs in the context of a sequence of vectors \( x^{(t)} \) with \( t = 1, \ldots, \tau \) as input.

Sometimes we are interested in only taking a single, fixed sized vector \( x \) as input, that generates the \( y \) sequence.

Some common ways to provide an extra input to an RNN are:

- As an extra input at each time step
- As the initial state \( h^{(0)} \)
- Both
Design Patterns: Fixed vector as input

- The first option (extra input at each time step) is the most common:
The first option (extra input at each time step) is the most common:
The first option (extra input at each time step) is the most common:

Maps a fixed vector $x$ into a distribution over sequences $Y$ ($x^T R$ effectively is a new bias parameter for each hidden unit)
Application: Caption Generation

Caption Generation

- Man in black shirt is playing guitar.
- Construction worker in orange safety vest is working on road.
- Two young girls are playing with lego toy.
- Boy is doing backflip on wakeboard.
Design Patterns: Bidirectional RNNs

- RNNs considered till now, all have a causal structure: state at time $t$ only captures information from the past $x^{(1)}, \ldots, x^{(t-1)}$.
RNNs considered till now, all have a causal structure: state at time $t$ only captures information from the past $x^{(1)}, \ldots, x^{(t-1)}$.

Sometimes we are interested in an output $y^{(t)}$ which may depend on the whole input sequence.
RNNs considered till now, all have a causal structure: state at time $t$ only captures information from the past $x^{(1)}, \ldots, x^{(t-1)}$

Sometimes we are interested in an output $y^{(t)}$ which may depend on the whole input sequence.

Example: Interpretation of a current sound as a phoneme may depend on the next few due to co-articulation.
RNNs considered till now, all have a causal structure: state at time $t$ only captures information from the past $x^{(1)}, \ldots, x^{(t-1)}$

Sometimes we are interested in an output $y^{(t)}$ which may depend on the whole input sequence.

Example: Interpretation of a current sound as a phoneme may depend on the next few due to co-articulation.

Basically, in many cases we are interested in looking into the future as well as the past to disambiguate interpretations.
Design Patterns: Bidirectional RNNs

- RNNs considered till now, all have a causal structure: state at time $t$ only captures information from the past $x^{(1)}, \ldots, x^{(t-1)}$
- Sometimes we are interested in an output $y^{(t)}$ which may depend on the whole input sequence
- Example: Interpretation of a current sound as a phoneme may depend on the next few due to co-articulation
- Basically, in many cases we are interested in looking into the future as well as the past to disambiguate interpretations
- Bidirectional RNNs were introduced to address this need (Schuster and Paliwal, 1997), and have been used in handwriting recognition (Graves 2012, Graves and Schmidhuber 2009), speech recognition (Graves and Schmidhuber 2005) and bioinformatics (Baldi 1999)
Design Patterns: Bidirectional RNNs
How do we map input sequences to output sequences that are not necessarily of the same length?

Example: Input - Kérem jójtenek maskor mászhoz és különösen máshoz. Output - 'Please come rather at another time and to another person.'

Other example applications: Speech recognition, question answering etc.

The input to this RNN is called the context, we want to find a representation of the context $C$. $C$ could be a vector or a sequence that summarizes $X = \{x(1), ..., x(n)\}$. 

Lecture 11 Recurrent Neural Networks I CMSC 35246
How do we map input sequences to output sequences that are not necessarily of the same length?

Example: Input - Kérem jöjjenek máskor és különösen máshoz. Output - 'Please come rather at another time and to another person.'
How do we map input sequences to output sequences that are not necessarily of the same length?

Example: Input - Kérem jöjjenek máskor és különösen máshoz. Output - 'Please come rather at another time and to another person.'

Other example applications: Speech recognition, question answering etc.
How do we map input sequences to output sequences that are not necessarily of the same length?

Example: Input - Kérem jöjjenek máskor és különösen máshoz. Output - 'Please come rather at another time and to another person.'

Other example applications: Speech recognition, question answering etc.

The input to this RNN is called the context, we want to find a representation of the context $C$.
How do we map input sequences to output sequences that are not necessarily of the same length?

Example: Input - Kérem jöjjenek máskor és különösen máshoz. Output - 'Please come rather at another time and to another person.'

Other example applications: Speech recognition, question answering etc.

The input to this RNN is called the context, we want to find a representation of the context $C$

$C$ could be a vector or a sequence that summarizes $X = \{ x^{(1)}, \ldots, x^{(n_x)} \}$
Far more complicated mappings
In the context of Machine Trans. $C$ is called a thought vector.
Deep Recurrent Networks

- The computations in RNNs can be decomposed into three blocks of parameters/associated transformations:
Deep Recurrent Networks

- The computations in RNNs can be decomposed into three blocks of parameters/associated transformations:
  - Input to hidden state
Deep Recurrent Networks

- The computations in RNNs can be decomposed into three blocks of parameters/associated transformations:
  - Input to hidden state
  - Previous hidden state to the next

Introducing depth in each of these operations is advantageous (Graves et al. 2013, Pascanu et al. 2014). The intuition on why depth should be more useful is quite similar to that in deep feed-forward networks. Optimization can be made much harder, but can be mitigated by tricks such as introducing skip connections.
Deep Recurrent Networks

The computations in RNNs can be decomposed into three blocks of parameters/associated transformations:

- Input to hidden state
- Previous hidden state to the next
- Hidden state to the output
Deep Recurrent Networks

- The computations in RNNs can be decomposed into three blocks of parameters/associated transformations:
  - Input to hidden state
  - Previous hidden state to the next
  - Hidden state to the output
- Each of these transforms till now were learned affine transformations followed by a fixed nonlinearity
Deep Recurrent Networks

- The computations in RNNs can be decomposed into three blocks of parameters/associated transformations:
  - Input to hidden state
  - Previous hidden state to the next
  - Hidden state to the output
- Each of these transforms till now were learned affine transformations followed by a fixed nonlinearity
- Introducing depth in each of these operations is advantageous (Graves et al. 2013, Pascanu et al. 2014)
Deep Recurrent Networks

- The computations in RNNs can be decomposed into three blocks of parameters/associated transformations:
  - Input to hidden state
  - Previous hidden state to the next
  - Hidden state to the output
- Each of these transforms till now were learned affine transformations followed by a fixed nonlinearity
- Introducing depth in each of these operations is advantageous (Graves et al. 2013, Pascanu et al. 2014)
- The intuition on why depth should be more useful is quite similar to that in deep feed-forward networks
The computations in RNNs can be decomposed into three blocks of parameters/associated transformations:

- Input to hidden state
- Previous hidden state to the next
- Hidden state to the output

Each of these transforms till now were learned affine transformations followed by a fixed nonlinearity.

Introducing depth in each of these operations is advantageous (Graves et al. 2013, Pascanu et al. 2014).

The intuition on why depth should be more useful is quite similar to that in deep feed-forward networks.

Optimization can be made much harder, but can be mitigated by tricks such as introducing *skip connections*.
Deep Recurrent Networks

(b) lengthens shortest paths linking different time steps, (c) mitigates this by introducing skip layers
Recursive Neural Networks

The computational graph is structured as a deep tree rather than as a chain in a RNN.
Recursive Neural Networks

- First introduced by Pollack (1990), used in Machine Reasoning by Bottou (2011)
Recursive Neural Networks

- First introduced by Pollack (1990), used in Machine Reasoning by Bottou (2011)
- Successfully used to process data structures as input to neural networks (Frasconi et al 1997), Natural Language Processing (Socher et al 2011) and Computer vision (Socher et al 2011)

Advantage: For sequences of length $\tau$, the number of compositions of nonlinear operations can be reduced from $\tau$ to $O(\log \tau)$

Choice of tree structure is not very clear
- A balanced binary tree, that does not depend on the structure of the data has been used in many applications
- Sometimes domain knowledge can be used: Parse trees given by a parser in NLP (Socher et al 2011)

The computation performed by each node need not be the usual neuron computation - it could instead be tensor operations etc (Socher et al 2013)
Recursive Neural Networks

- First introduced by Pollack (1990), used in Machine Reasoning by Bottou (2011)
- Successfully used to process data structures as input to neural networks (Frasconi et al. 1997), Natural Language Processing (Socher et al. 2011) and Computer vision (Socher et al. 2011)
- Advantage: For sequences of length $\tau$, the number of compositions of nonlinear operations can be reduced from $\tau$ to $O(\log \tau)$
Recursive Neural Networks

- First introduced by Pollack (1990), used in Machine Reasoning by Bottou (2011)
- Successfully used to process data structures as input to neural networks (Frasconi et al 1997), Natural Language Processing (Socher et al 2011) and Computer vision (Socher et al 2011)
- Advantage: For sequences of length $\tau$, the number of compositions of nonlinear operations can be reduced from $\tau$ to $O(\log \tau)$
- Choice of tree structure is not very clear
Recursive Neural Networks

- First introduced by Pollack (1990), used in Machine Reasoning by Bottou (2011)
- Successfully used to process data structures as input to neural networks (Frasconi et al 1997), Natural Language Processing (Socher et al 2011) and Computer vision (Socher et al 2011)
- Advantage: For sequences of length $\tau$, the number of compositions of nonlinear operations can be reduced from $\tau$ to $O(\log \tau)$
- Choice of tree structure is not very clear
  - A balanced binary tree, that does not depend on the structure of the data has been used in many applications
Recursive Neural Networks

- First introduced by Pollack (1990), used in Machine Reasoning by Bottou (2011)
- Successfully used to process data structures as input to neural networks (Frasconi et al. 1997), Natural Language Processing (Socher et al. 2011) and Computer vision (Socher et al. 2011)
- Advantage: For sequences of length $\tau$, the number of compositions of nonlinear operations can be reduced from $\tau$ to $O(\log \tau)$
- Choice of tree structure is not very clear
  - A balanced binary tree, that does not depend on the structure of the data has been used in many applications
  - Sometimes domain knowledge can be used: Parse trees given by a parser in NLP (Socher et al. 2011)
Recursive Neural Networks

- First introduced by Pollack (1990), used in Machine Reasoning by Bottou (2011)
- Successfully used to process data structures as input to neural networks (Frasconi et al 1997), Natural Language Processing (Socher et al 2011) and Computer vision (Socher et al 2011)
- Advantage: For sequences of length $\tau$, the number of compositions of nonlinear operations can be reduced from $\tau$ to $O(\log \tau)$
- Choice of tree structure is not very clear
  - A balanced binary tree, that does not depend on the structure of the data has been used in many applications
  - Sometimes domain knowledge can be used: Parse trees given by a parser in NLP (Socher et al 2011)
- The computation performed by each node need not be the usual neuron computation - it could instead be tensor operations etc (Socher et al 2013)
Long-Term Dependencies
Challenge of Long-Term Dependencies

- Basic problem: Gradients propagated over many stages tend to vanish (most of the time) or explode (relatively rarely)
Challenge of Long-Term Dependencies

- Basic problem: Gradients propagated over many stages tend to vanish (most of the time) or explode (relatively rarely)
- Difficulty with long term interactions (involving multiplication of many jacobians) arises due to exponentially smaller weights, compared to short term interactions
Challenge of Long-Term Dependencies

- Basic problem: Gradients propagated over many stages tend to vanish (most of the time) or explode (relatively rarely).
- Difficulty with long term interactions (involving multiplication of many jacobians) arises due to exponentially smaller weights, compared to short term interactions.
- The problem was first analyzed by Hochreiter and Schmidhuber 1991 and Bengio et al 1993.
Challenge of Long-Term Dependencies

- Recurrent Networks involve the composition of the same function multiple times, once per time step.
Challenge of Long-Term Dependencies

- Recurrent Networks involve the composition of the same function multiple times, once per time step.
- The function composition in RNNs somewhat resembles matrix multiplication.
Challenge of Long-Term Dependencies

- Recurrent Networks involve the composition of the same function multiple times, once per time step.
- The function composition in RNNs somewhat resembles matrix multiplication.
- Consider the recurrence relationship:

\[ h(t) = W^T h(t-1) \]
Challenge of Long-Term Dependencies

- Recurrent Networks involve the composition of the same function multiple times, once per time step.
- The function composition in RNNs somewhat resembles matrix multiplication.
- Consider the recurrence relationship:
  \[ h^{(t)} = W^T h^{(t-1)} \]

- This could be thought of as a very simple recurrent neural network without a nonlinear activation and lacking \( x \).
Challenge of Long-Term Dependencies

- Recurrent Networks involve the composition of the same function multiple times, once per time step.
- The function composition in RNNs somewhat resembles matrix multiplication.
- Consider the recurrence relationship:
  \[
  h^{(t)} = W^T h^{(t-1)}
  \]

  - This could be thought of as a very simple recurrent neural network without a nonlinear activation and lacking \( x \).
  - This recurrence essentially describes the power method and can be written as:
    \[
    h^{(t)} = (W^t)^T h^{(0)}
    \]
Challenge of Long-Term Dependencies

- If $W$ admits a decomposition $W = Q\Lambda Q^T$ with orthogonal $Q$
Challenge of Long-Term Dependencies

- If $W$ admits a decomposition $W = Q\Lambda Q^T$ with orthogonal $Q$
- The recurrence becomes:

$$h(t) = (W^t)^T h(0) = Q^T \Lambda^t Q h(0)$$
Challenge of Long-Term Dependencies

- If $W$ admits a decomposition $W = Q\Lambda Q^T$ with orthogonal $Q$
- The recurrence becomes:

$$h(t) = (W^t)^T h(0) = Q^T \Lambda^t Q h(0)$$

- Eigenvalues are raised to $t$: Quickly decay to zero or explode
Challenge of Long-Term Dependencies

- If $W$ admits a decomposition $W = Q\Lambda Q^T$ with orthogonal $Q$
- The recurrence becomes:

$$h(t) = (W^t)^T h(0) = Q^T \Lambda^t Q h(0)$$

- Eigenvalues are raised to $t$: Quickly decay to zero or explode
- Problem particular to RNNs
Solution 1: Echo State Networks

Idea: Set the recurrent weights such that they do a good job of capturing past history and learn only the output weights.
Solution 1: Echo State Networks

- **Idea:** Set the recurrent weights such that they do a *good job* of capturing past history and learn only the output weights.
- **Methods:** Echo State Machines, Liquid State Machines
Solution 1: Echo State Networks

- **Idea:** Set the recurrent weights such that they do a *good job* of capturing past history and learn only the output weights.
- **Methods:** Echo State Machines, Liquid State Machines
- **The general methodology is called reservoir computing.**
Solution 1: Echo State Networks

- **Idea:** Set the recurrent weights such that they do a *good job* of capturing past history and learn only the output weights
- **Methods:** Echo State Machines, Liquid State Machines
- The general methodology is called reservoir computing
- How to choose the recurrent weights?
Original idea: Choose recurrent weights such that the hidden-to-hidden transition Jacobian has eigenvalues close to 1
Echo State Networks

**Original idea:** Choose recurrent weights such that the hidden-to-hidden transition Jacobian has eigenvalues close to 1.

In particular we pay attention to the spectral radius of $J_t$. 
**Echo State Networks**

- **Original idea:** Choose recurrent weights such that the hidden-to-hidden transition Jacobian has eigenvalues close to 1
- In particular we pay attention to the spectral radius of $J_t$
- Consider gradient $g$, after one step of backpropagation it would be $Jg$ and after $n$ steps it would be $J^n g$
**Echo State Networks**

- **Original idea:** Choose recurrent weights such that the hidden-to-hidden transition Jacobian has eigenvalues close to 1.
- In particular we pay attention to the spectral radius of $J_t$.
- Consider gradient $g$, after one step of backpropagation it would be $Jg$ and after $n$ steps it would be $J^n g$.
- Now consider a perturbed version of $g$ i.e. $g + \delta v$, after $n$ steps we will have $J^n (g + \delta v)$.
Echo State Networks

- **Original idea:** Choose recurrent weights such that the hidden-to-hidden transition Jacobian has eigenvalues close to 1.
- In particular we pay attention to the spectral radius of $J_t$.
- Consider gradient $g$, after one step of backpropagation it would be $Jg$ and after $n$ steps it would be $J^n g$.
- Now consider a perturbed version of $g$ i.e. $g + \delta v$, after $n$ steps we will have $J^n (g + \delta v)$.
- Infact, the separation is exactly $\delta |\lambda|^n$. 
Echo State Networks

**Original idea:** Choose recurrent weights such that the hidden-to-hidden transition Jacobian has eigenvalues close to 1.

- In particular we pay attention to the spectral radius of $J_t$.
- Consider gradient $g$, after one step of backpropagation it would be $Jg$ and after $n$ steps it would be $J^n g$.
- Now consider a perturbed version of $g$ i.e. $g + \delta v$, after $n$ steps we will have $J^n (g + \delta v)$.
- Infact, the separation is exactly $\delta |\lambda|^n$.
- When $|\lambda > 1|$, $\delta |\lambda|^n$ grows exponentially large and vice-versa.
Echo State Networks

- For a vector $h$, when a linear map $W$ always shrinks $h$, the mapping is said to be contractive.
Echo State Networks

- For a vector $h$, when a linear map $W$ always shrinks $h$, the mapping is said to be contractive.
- The strategy of echo state networks is to make use of this intuition.
Echo State Networks

- For a vector $h$, when a linear map $W$ always shrinks $h$, the mapping is said to be contractive.
- The strategy of echo state networks is to make use of this intuition.
- The Jacobian is chosen such that the spectral radius corresponds to stable dynamics.
Other Ideas

- Skip Connections
Other Ideas

- Skip Connections
- Leaky Units
Long Short Term Memory

$h_{t-1} \rightarrow \tanh$

$x_t$

$h_t = \tanh(W h_{t-1} + U x_t)$
Long Short Term Memory

\[
\tilde{c}_t = \tanh(W h_{t-1} + U x_t)
\]

\[
c_t = c_{t-1} + \tilde{c}_t
\]
Long Short Term Memory

\[ f_t = \sigma(W_f h_{t-1} + U_f x_t) \]

\[ \tilde{c}_t = \tanh(W h_{t-1} + U x_t) \]

\[ c_t = f_t \odot c_{t-1} + \tilde{c}_t \]
Long Short Term Memory

\[ f_t = \sigma(W_f h_{t-1} + U_f x_t) \]
\[ i_t = \sigma(W_i h_{t-1} + U_i x_t) \]
\[ \tilde{c}_t = \tanh(W h_{t-1} + U x_t) \]
\[ c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \]
Long Short Term Memory

\[ \tilde{c}_t = \tanh(W h_{t-1} + U x_t) \]
\[ c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \]
\[ h_t = o_t \odot \tanh(c_t) \]

\[ f_t = \sigma(W_f h_{t-1} + U_f x_t) \]
\[ i_t = \sigma(W_i h_{t-1} + U_i x_t) \]
\[ o_t = \sigma(W_o h_{t-1} + U_o x_t) \]
Gated Recurrent Unit

- Let $\tilde{h}_t = \tanh(W h_{t-1} + U x_t)$ and $h_t = \tilde{h}_t$
Gated Recurrent Unit

- Let \( \tilde{h}_t = \tanh(W h_{t-1} + U x_t) \) and \( h_t = \tilde{h}_t \)
- Reset gate: \( r_t = \sigma(W_r h_{t-1} + U_r x_t) \)
Gated Recurrent Unit

Let $\tilde{h}_t = \tanh(W h_{t-1} + U x_t)$ and $h_t = \tilde{h}_t$

Reset gate: $r_t = \sigma(W_r h_{t-1} + U_r x_t)$

New $\tilde{h}_t = \tanh(W (r_t \odot h_{t-1}) + U x_t)$
Gated Recurrent Unit

- Let $\tilde{h}_t = \tanh(W h_{t-1} + U x_t)$ and $h_t = \tilde{h}_t$
- Reset gate: $r_t = \sigma(W_r h_{t-1} + U_r x_t)$
- New $\tilde{h}_t = \tanh(W (r_t \odot h_{t-1}) + U x_t)$
- Find: $z_t = \sigma(W_z h_{t-1} + U_z x_t)$

Comes from attempting to factor LSTM and reduce gates.
Gated Recurrent Unit

- Let $\tilde{h}_t = \tanh(W h_{t-1} + U x_t)$ and $h_t = \tilde{h}_t$
- Reset gate: $r_t = \sigma(W_r h_{t-1} + U_r x_t)$
- New $\tilde{h}_t = \tanh(W (r_t \circ h_{t-1}) + U x_t)$
- Find: $z_t = \sigma(W_z h_{t-1} + U_z x_t)$
- Update $h_t = z_t \circ \tilde{h}_t$
Let $\tilde{h}_t = \tanh(W h_{t-1} + U x_t)$ and $h_t = \tilde{h}_t$

- Reset gate: $r_t = \sigma(W_r h_{t-1} + U_r x_t)$
- New $\tilde{h}_t = \tanh(W (r_t \odot h_{t-1}) + U x_t)$
- Find: $z_t = \sigma(W_z h_{t-1} + U_z x_t)$
- Update $h_t = z_t \odot \tilde{h}_t$
- Finally: $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$
Gated Recurrent Unit

- Let $\tilde{h}_t = \tanh(W h_{t-1} + U x_t)$ and $h_t = \tilde{h}_t$
- Reset gate: $r_t = \sigma(W_r h_{t-1} + U_r x_t)$
- New $\tilde{h}_t = \tanh(W (r_t \odot h_{t-1}) + U x_t)$
- Find: $z_t = \sigma(W_z h_{t-1} + U_z x_t)$
- Update $h_t = z_t \odot \tilde{h}_t$
- Finally: $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$
- Comes from attempting to factor LSTM and reduce gates