Objectives

1. Understanding why we need RNNs
2. Understanding the working principles of RNNs
3. Implementation of LSTMs in PyTorch
4. Variants of RNNs
5. RNNs from a Perspective of Graphical Models
6. Normalization/Dropout in RNNs
Why Recurrent Neural Network (RNNs)?

- RNNs learn the sequential characteristics in data inputs and makes predictions of the next possible outcomes.
- RNNs have the ability to process temporal information present in sequential data.
- RNNs have hidden states which act as the memory of the neural network which remembers information on data sequence.
Sequential Data

- Consecutive data inputs which are dependent on each other
- Data input n is dependent on n-1 and n-1 dependent on n-2...
- Example:
  - Text: (Where are you off to?)
  - Audio/speech
  - Video
  - Time Series data (stock price, weather data)
Application of RNN

- Intelligent stock prediction systems
- Machine translation
- Speech Recognition systems
- Language modeling and Text generation
- Video Tagging
- Image captioning
- Text Summarization

**Software applications**: Google translate, Trading bots, Siri, Cortana, voice search
Data Modeling

Types of Recurrent Neural Network

**one to one**

![Diagram of Image Classification](ref)

**one to many**

"man in black shirt is playing guitar."

![Diagram of Image Captioning](ref)
Data Modeling

Types of Recurrent Neural Network

**many to one**

**many to many**

**many to many**

*Sentiment Analysis (Movie Review)*
The Batman (2022) is everything a superhero movie should be. *(Positive)*

*Machine Translation*
“How are you?” -> “எப்படி இருக்கிறீர்கள்?”

*Object Tracking in videos*
Recurrent Neural Networks

- Feeds the output of previous layer without activation to the input of next layer
- Looping Network (output of network depend on previous input of the sequence)
- Parameter sharing across timesteps
- Derivatives is aggregated across all the timesteps
- Backpropagation through time (BPTT)

A, B and C are the parameters
Recurrent Neural Networks

At any given time t, the current input is a combination of input at x(t) and the output from the previous hidden layer h(t-1)
# Problems with RNN

## Vanishing gradient

Gradient back propagated through time becomes too small and loose information.

**Potential solutions:**
- Weight initialization
- Choosing the right activation
- Long Short-Term Memory Networks (LSTMs)

## Exploding gradient

Gradient tends to grow exponentially instead of decaying and cannot be contained.

**Potential solutions:**
- Identity Initialization
- Truncated backpropagation
- Gradient clipping

To avoid these gradient problems we need to decide the amount of information we would need to retain for prediction.
Long Short-Term Memory (LSTMs)

- Capable of retaining information over a long period of time
- The most popular and efficient way of dealing with gradient problems
- LSTMs have chain-like structure repeating each cell
- Each LSTM cell have a defined structure depending on the variant
- LSTM Variants: -> LSTM Classic, LSTM Peepole connection
- Gated Recurrent Unit (GRU)
Classic LSTM

### Gates

**Forget gate**
\[ f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \]

**Input gate**
\[ i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \]

**Output gate**
\[ o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \]

### Variables

\[ \tilde{C}_t = \text{tanh}(W_C \cdot [h_{t-1}, x_t] + b_C) \]
\[ C_t = \sigma(f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t) \]
\[ h_t = o_t \cdot \text{tanh}(C_t) \]

(Gers and Schmidhuber 2000: Recurrent Nets that Time and Count)
Peephole LSTM

Gates

**Forget gate**
\[ f_t = \sigma (W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f) \]

**Input gate**
\[ i_t = \sigma (W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i) \]

**Output gate**
\[ o_t = \sigma (W_o \cdot [C_t, h_{t-1}, x_t] + b_o) \]

Variables

\[ \tilde{C}_t = \tanh (W_C \cdot [h_{t-1}, x_t] + b_C) \]
\[ C_t = \sigma (f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t) \]
\[ h_t = o_t \cdot \tanh (C_t) \]
LSTM PyTorch Implementation

```python
input_size = 1  # The number of variables in your sequence data.
n_hidden = 100  # The number of hidden nodes in the LSTM layer.
n_layers = 2   # The total number of LSTM layers to stack.
out_size = 1   # The size of the output you desire from your RNN.

lstm = nn.LSTM(input_size, n_hidden, n_layers, batch_first=True)
linear = nn.Linear(n_hidden, 1)
```

# 1. network input shape: (batch_size, seq_length, num_features)
# 2. LSTM output shape: (batch_size, seq_length, hidden_size)
# 3. Linear input shape: (batch_size * seq_length, hidden_size)
# 4. Linear output: (batch_size * seq_length, out_size)
Questions:
1. What are weight_ih and weight_hh?
2. How to interpret the dimensions?
3. Which version of LSTM is this?
4. How should you use initialization (e.g. Xavier, Kaiming)?
Questions:
1. What are weight_ih and weight_hh? Input weights and hidden weights
2. How to interpret the dimensions? Input, forget, cell, and output weights stacked (reference)
3. Which version of LSTM is this? Wikipedia version (no peephole connection)
4. How should you use initialization (e.g. Xavier, Kaiming)? We initialize each one of four (three if GRU) matrices separately
Performance per LSTM Component

CIFG: GRU, NP: No peepholes, FGR: Full gate recurrence, NOG: No output gate, NIG: No input gate, NFG: No forget gate, NIAF: No input activation function, NOAF: No output activation function

(Greff et al. 2017: LSTM: A Search Space Odyssey)
## Performance per LSTM Component

<table>
<thead>
<tr>
<th>Arch.</th>
<th>Arith.</th>
<th>XML</th>
<th>PTB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tanh</td>
<td>0.29493</td>
<td>0.32050</td>
<td>0.08782</td>
</tr>
<tr>
<td>LSTM</td>
<td>0.89228</td>
<td>0.42470</td>
<td>0.08912</td>
</tr>
<tr>
<td>LSTM-f</td>
<td>0.29292</td>
<td>0.23356</td>
<td>0.08808</td>
</tr>
<tr>
<td>LSTM-i</td>
<td>0.75109</td>
<td>0.41371</td>
<td>0.08662</td>
</tr>
<tr>
<td>LSTM-o</td>
<td>0.86747</td>
<td>0.42117</td>
<td>0.08933</td>
</tr>
<tr>
<td>LSTM-b</td>
<td>0.90163</td>
<td>0.44434</td>
<td>0.08952</td>
</tr>
<tr>
<td>GRU</td>
<td>0.89565</td>
<td>0.45963</td>
<td>0.09069</td>
</tr>
<tr>
<td>MUT1</td>
<td>0.92135</td>
<td>0.47483</td>
<td>0.08968</td>
</tr>
<tr>
<td>MUT2</td>
<td>0.89735</td>
<td>0.47324</td>
<td>0.09036</td>
</tr>
<tr>
<td>MUT3</td>
<td>0.90728</td>
<td>0.46478</td>
<td><strong>0.09161</strong></td>
</tr>
</tbody>
</table>

(Jozefowicz et al. 2015: *An Empirical Exploration of Recurrent Network Architectures*)
Gated Recurrent Unit (GRU)

**Gates**

- **reset gate**
  \[ r_t = \sigma (W_r \cdot [h_{t-1}, x_t]) \]

- **update gate**
  \[ z_t = \sigma (W_z \cdot [h_{t-1}, x_t]) \]

**Variables**

\[ \tilde{h}_t = \tanh (W \cdot [r_t \cdot h_{t-1}, x_t]) \]
\[ h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t \]
Actual Network with BRNNs
Recall: Hidden Markov Models (HMMs)
- A sequence of hidden variables $h_i$ and a sequence of observations $x_i$
- Conditional independence
- The hidden variable $h$ is discrete

Was very popular for sequential tasks
RNNs: A Perspective from Graphical Models

- State Space Model (SSMs): “continuous” version of HMMs
  - More complex than it sounds like
  - Implications: different inference algorithms

- A specific instance of SSMs: linear dynamical system
  - \( h_t = A h_{t-1} + B w_{t-1} \) (Transition)
  - \( X_t = C h_t + v_t \) (Observation)
  - \( A \) and \( C \) are two matrices
  - \( w_t \) and \( v_t \) are two noise terms
RNNs: A Perspective from Graphical Models

- State Space Model
  - $h_t = A h_{t-1} + B w_t$ (Transition)
  - $X_t = C h_t + v_t$ (Observation)
- An estimator of the true hidden variables (Kalman Filter)
  - $h_t = A' h_{t-1} + B' X_t$
- What about a non-linear estimator?
  - $h_t = f(h_{t-1}, X_t)$
  - We get RNN!
RNNs + Normalization?

- Applying BN to RNNs
  - The statistics could be different for different time steps
  - Typically does not work well

- Recurrent Batch Normalization [1]
  - Separate BN for recurrent term and input term
  - Separate statistics for different time step
    - But what if the length of test data is longer than all of the training data?
  - Initialization of gain matters

- Layer Normalization [2]
  - Proposed for RNNs but still not very common for RNNs
  - Widely used in other scenarios, e.g. Transformers


RNNs + Dropout?

- `dropout` in PyTorch nn.LSTM
  - No effect if `num_layers=1`
  - Most common
- Two Types of Connections in RNNs
  - Layer-to-Layer
  - Hidden-to-Hidden
Variational RNNs, a.k.a. Locked Dropout

- Locked Dropout [1]
  - Key idea: keep the dropout mask fixed
  - Note: the implementation in torchnlp is slightly different
    - Only applied to layer-to-layer connection

---

Weight-dropped LSTM

- Weight-dropped LSTM [1]
  - Essentially, “DropConnect”. Dropout applied to weight matrices

Language Model Demo: Shakespeare