Recurrent Neural Networks (RNNs) + Module-based Automatic Differentiation
RECURRENT NEURAL NETWORKS
Dataset for Supervised Part-of-Speech (POS) Tagging

Data: \[ \mathcal{D} = \{ x^{(n)}, y^{(n)} \}_{n=1}^{N} \]

Sample 1:
- time
- flies
- like
- an
- arrow

Sample 2:
- time
- flies
- like
- an
- arrow

Sample 3:
- flies
- fly
- with
- their
- wings

Sample 4:
- with
- time
- you
- will
- see
Dataset for Supervised Handwriting Recognition

Data: \[ D = \{ x^{(n)}, y^{(n)} \}_{n=1}^N \]

Sample 1:

\[
\begin{array}{cccccccc}
\text{u} & \text{n} & \text{e} & \text{x} & \text{p} & \text{e} & \text{c} & \text{t} & \text{e} & \text{d} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{U} & \text{h} & \text{e} & \text{x} & \text{p} & \text{e} & \text{c} & \text{t} & \text{e} \text{d} \\
\end{array}
\]

Sample 2:

\[
\begin{array}{cccccccc}
\text{v} & \text{o} & \text{l} & \text{c} & \text{a} & \text{n} & \text{i} & \text{c} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{y} & \text{d} & \text{i} & \text{c} & \text{a} & \text{n} & \text{i} & \text{c} \\
\end{array}
\]

Sample 3:

\[
\begin{array}{cccccccc}
\text{e} & \text{m} & \text{b} & \text{r} & \text{a} & \text{c} & \text{e} & \text{s} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{E} & \text{m} & \text{b} & \text{r} & \text{a} & \text{c} & \text{e} \text{s} \\
\end{array}
\]

Figures from (Chatzis & Demiris, 2013)
Dataset for Supervised Phoneme (Speech) Recognition

Data: \( \mathcal{D} = \{ x^{(n)}, y^{(n)} \}_{n=1}^{N} \)

Sample 1:

- h#
- dh
- ih
- s
- w
- uh
- z
- iy
- z
- iy

Sample 2:

- f
- ao
- r
- ah
- s
- s
- s
- h#

Figures from (Jansen & Niyogi, 2013)
Question 1: How could we apply the neural networks we’ve seen so far (which expect fixed size input/output) to a prediction task with variable length input/output?
Question 1: How could we apply the neural networks we’ve seen so far (which expect fixed size input/output) to a prediction task with variable length input/output?
**Time Series Data**

**Question 2**: How could we incorporate context (e.g. words to the left/right, or tags to the left/right) into our solution?

![Diagram showing context incorporation]

**Multiple Choice:** Working left-to-right, use features of...

<table>
<thead>
<tr>
<th></th>
<th>$x_{i-1}$</th>
<th>$x_i$</th>
<th>$x_{i+1}$</th>
<th>$y_{i-1}$</th>
<th>$y_i$</th>
<th>$y_{i+1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>E</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>F</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>G</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>H</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Recurrent Neural Networks (RNNs)

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathbb{R}^I \)

hidden units: \( h = (h_1, h_2, \ldots, h_T), h_i \in \mathbb{R}^J \)

outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathbb{R}^K \)

nonlinearity: \( \mathcal{H} \in \{ \text{sig}, \text{relu}, \text{elu} \} \)

Definition of the RNN:

\[
\begin{align*}
  h_t &= \mathcal{H} \left( W_{xh} x_t + W_{hh} h_{t-1} + b_h \right) \\
  y_t &= W_{hy} h_t + b_y
\end{align*}
\]
Recurrent Neural Networks (RNNs)

inputs: \( \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( \mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J \)

outputs: \( \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Definition of the RNN:
\[
h_t = \mathcal{H} (W_{xh} x_t + W_{hh} h_{t-1} + b_h)
\]
\[
y_t = W_{hy} h_t + b_y
\]

This form of RNN is called an Elman Network
Recurrent Neural Networks (RNNs)

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathbb{R}^I \)

hidden units: \( h = (h_1, h_2, \ldots, h_T), h_i \in \mathbb{R}^J \)

outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathbb{R}^K \)

nonlinearity: \( \mathcal{H} \)

Definition of the RNN:

\[
\begin{align*}
    h_t &= \mathcal{H} (W_{xh} x_t + W_{hh} h_{t-1} + b_h) \\
    y_t &= W_{hy} h_t + b_y
\end{align*}
\]

- If \( T=1 \), then we have a standard feed-forward neural net with one hidden layer
- All of the deep nets from last lecture required fixed size inputs/outputs
1. Given training data:
\[
\{ x_i, y_i \}_{i=1}^{N}
\]

2. Choose each of these:
- Decision function
  \[ \hat{y} = f_\theta(x_i) \]
- Loss function
  \[ \ell(\hat{y}, y_i) \in \mathbb{R} \]

3. Define goal:
\[
\theta^* = \arg \min_{\theta} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i)
\]

4. Train with SGD:
(take small steps opposite the gradient)
\[
\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla \ell(f_\theta(x_i), y_i)
\]
1. Given training data:
2. Choose each of these:
   - Decision function
   - Loss function
3. Define goal:
4. Train with SGD:
   (take small steps opposite the gradient)
   - We’ll just need a method of computing the gradient efficiently
   - Let’s use Backpropagation Through Time...

• Recurrent Neural Networks (RNNs) provide another form of decision function
• An RNN is just another differential function
Recurrent Neural Networks (RNNs)

inputs: \( \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( \mathbf{h} = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J \)

outputs: \( \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Definition of the RNN:
\[
\begin{align*}
    h_t &= \mathcal{H} \left( W_{xh} x_t + W_{hh} h_{t-1} + b_h \right) \\
    y_t &= W_{hy} h_t + b_y
\end{align*}
\]
Recurrent Neural Networks (RNNs)

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)
hidden units: \( h = (h_1, h_2, \ldots, h_T), h_i \in \mathcal{R}^J \)
outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)
nonlinearity: \( \mathcal{H} \)

Definition of the RNN:
\[
\begin{align*}
    h_t &= \mathcal{H}(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\
    y_t &= W_{hy}h_t + b_y
\end{align*}
\]

- By unrolling the RNN through time, we can share parameters and accommodate arbitrary length input/output pairs
- Applications: time-series data such as sentences, speech, stock-market, signal data, etc.

\[
\begin{align*}
    y_t &\quad \text{output} \\
    h &\quad \text{hidden state} \\
    x_t &\quad \text{input at time step } t
\end{align*}
\]
Background: Backprop through time

Recurrent neural network:

BPTT:
1. Unroll the computation over time
2. Run backprop through the resulting feedforward network

(Robinson & Fallside, 1987)
(Werbos, 1988)
(Mozer, 1995)
inputs: \( \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( \overrightarrow{\mathbf{h}} \) and \( \overleftarrow{\mathbf{h}} \)

outputs: \( \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Recursive Definition:

\[
\begin{align*}
\overrightarrow{h}_t &= \mathcal{H} \left( W_{x h} \overrightarrow{x}_t + W_{h h} \overrightarrow{h}_{t-1} + b_h \right) \\
\overleftarrow{h}_t &= \mathcal{H} \left( W_{x h} \overleftarrow{x}_t + W_{h h} \overleftarrow{h}_{t+1} + b_h \right) \\
y_t &= W_{h y} \overrightarrow{h}_t + W_{h y} \overleftarrow{h}_t + b_y
\end{align*}
\]
Bidirectional RNN

inputs: \( \mathbf{x} = (x_1, x_2, \ldots, x_T), x_i \in \mathbb{R}^I \)

hidden units: \( \mathbf{\hat{h}} \) and \( \mathbf{\hat{h}} \)

outputs: \( \mathbf{y} = (y_1, y_2, \ldots, y_T), y_i \in \mathbb{R}^K \)

nonlinearity: \( \mathcal{H} \)

Recursive Definition:

\[
\begin{align*}
\mathbf{\hat{h}}_t &= \mathcal{H} \left( W_{x_{\hat{h}}} x_t + W_{\hat{h}_{\hat{h}}} \mathbf{\hat{h}}_{t-1} + b_{\hat{h}} \right) \\
\mathbf{\hat{h}}_t &= \mathcal{H} \left( W_{\hat{x}_{\hat{h}}} x_t + W_{\hat{h}_{\hat{h}}} \mathbf{\hat{h}}_{t+1} + b_{\hat{h}} \right) \\
y_t &= W_{\hat{h}_{y}} \mathbf{\hat{h}}_t + W_{h_{y}} \mathbf{\hat{h}}_t + b_y
\end{align*}
\]
Bidirectional RNN

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

hidden units: \( \vec{h} \) and \( \hat{h} \)

outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

Recursive Definition:
\[
\vec{h}_t = \mathcal{H} \left( W_{x \vec{h}} x_t + W_{\vec{h} \vec{h}} \vec{h}_{t-1} + b_{\vec{h}} \right)
\]
\[
\hat{h}_t = \mathcal{H} \left( W_{x \hat{h}} x_t + W_{\hat{h} \hat{h}} \hat{h}_{t+1} + b_{\hat{h}} \right)
\]
\[
y_t = W_{h_y \vec{h}} \vec{h}_t + W_{h_y \hat{h}} \hat{h}_t + b_y
\]

Is there an analogy to some other recursive algorithm(s) we know?
Deep RNNs

inputs: $x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I$
outputs: $y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K$
nonlinearity: $\mathcal{H}$

Recursive Definition:

$$h^n_t = \mathcal{H} \left( W_{h^{n-1}h^n}h_{t-1}^{n-1} + W_{h^{n}h^n}h_t^{n-1} + b^n_n \right)$$

$$y_t = W_h y_t^N + b_y$$

Figure from (Graves et al., 2013)
Deep Bidirectional RNNs

inputs: \( x = (x_1, x_2, \ldots, x_T), x_i \in \mathcal{R}^I \)

outputs: \( y = (y_1, y_2, \ldots, y_T), y_i \in \mathcal{R}^K \)

nonlinearity: \( \mathcal{H} \)

- Notice that the upper level hidden units have input from two previous layers (i.e. wider input)
- Likewise for the output layer
- What analogy can we draw to DNNs, DBNs, DBMs?

Figure from (Graves et al., 2013)
Long Short-Term Memory (LSTM)

Motivation:
- Standard RNNs have trouble learning long distance dependencies
- LSTMs combat this issue
Long Short-Term Memory (LSTM)

Motivation:
• Vanishing gradient problem for Standard RNNs
• Figure shows sensitivity (darker = more sensitive) to the input at time $t=1$

Figure from (Graves, 2012)
Long Short-Term Memory (LSTM)

Motivation:
• LSTM units have a rich internal structure
• The various “gates” determine the propagation of information and can choose to “remember” or “forget” information

Figure from (Graves, 2012)
Long Short-Term Memory (LSTM)
Long Short-Term Memory (LSTM)

- **Input gate**: masks out the standard RNN inputs
- **Forget gate**: masks out the previous cell
- **Cell**: stores the input/forget mixture
- **Output gate**: masks out the values of the next hidden

\[
i_t = \sigma (W_{xi} x_t + W_{hi} h_{t-1} + W_{ci} c_{t-1} + b_i)
\]
\[
f_t = \sigma (W_{xf} x_t + W_{hf} h_{t-1} + W_{cf} c_{t-1} + b_f)
\]
\[
c_t = f_t c_{t-1} + i_t \tanh (W_{xc} x_t + W_{hc} h_{t-1} + b_c)
\]
\[
o_t = \sigma (W_{xo} x_t + W_{ho} h_{t-1} + W_{co} c_t + b_o)
\]
\[
h_t = o_t \tanh(c_t)
\]

Figure from (Graves et al., 2013)
Vector only receives input from element cell where memory cells (e.g. are given in Section 7. y is usually an elementwise application of a sigmoid t = \text{sigmoid}(W_t x + b_t).}

2. NETWORK ARCHITECTURE

x_t \rightarrow h_t \rightarrow y_{t+1}

1. Deep bidirectional LSTM receives input from both the forward and backward layers at respectively computed from h_t = \text{tanh}(W_t x + b_t) y_t = \text{sigmoid}(W_t x + b_t) \text{for } t = 1, \ldots, T

Long Short-Term Memory (LSTM)

\begin{align*}
    i_t &= \sigma(x_t W_t i + h_{t-1} W_t i + b_t) \\
    f_t &= \sigma(x_t W_t f + h_{t-1} W_t f + b_t) \\
    o_t &= \sigma(x_t W_t o + h_{t-1} W_t o + b_t) \\
    c_t &= f_t c_{t-1} + i_t \text{tanh}(x_t W_t c) \\
    h_t &= o_t \text{tanh}(c_t) \\
    y_t &= \sigma(h_t W_t y + b_t)
\end{align*}

Long Short-Term Memory Cell

Deep bidirectional RNNs can be implemented by replacing the forward and backward layer from \( W_t \) with \( W_t \) and \( W_t \) respectively. The network outputs the forward and backward vector sequence \( h_t \) and \( h_t \) are iteratively computed from

Assuming the same hidden layer function is used for all layers in the stack, the hidden vector sequences \( h_t \) and \( h_t \) are respectively the forget gate \( f_t \) and output \( o_t \) for the level below. If LSTM is used for the hidden layers we get

The weight matrices from the cell to gate (e.g. input-hidden weight matrix), the is the hidden layer function. However we have found that the Long Short-Term Memory cell only able to make use of previous context. In speech recognition, this can be created by stacking multiple RNN hidden layers.

One shortcoming of conventional RNNs is that they are deep bidirectional LSTM, as illustrated in Fig. 4.

Long Short-Term Memory (LSTM)

\begin{align*}
    i_t &= \sigma(x_t W_t i + h_{t-1} W_t i + b_t) \\
    f_t &= \sigma(x_t W_t f + h_{t-1} W_t f + b_t) \\
    o_t &= \sigma(x_t W_t o + h_{t-1} W_t o + b_t) \\
    c_t &= f_t c_{t-1} + i_t \text{tanh}(x_t W_t c) \\
    h_t &= o_t \text{tanh}(c_t) \\
    y_t &= \sigma(h_t W_t y + b_t)
\end{align*}

Deep bidirectional LSTM, as illustrated in Fig. 4.

Long Short-Term Memory (LSTM)

\begin{align*}
    i_t &= \sigma(x_t W_t i + h_{t-1} W_t i + b_t) \\
    f_t &= \sigma(x_t W_t f + h_{t-1} W_t f + b_t) \\
    o_t &= \sigma(x_t W_t o + h_{t-1} W_t o + b_t) \\
    c_t &= f_t c_{t-1} + i_t \text{tanh}(x_t W_t c) \\
    h_t &= o_t \text{tanh}(c_t) \\
    y_t &= \sigma(h_t W_t y + b_t)
\end{align*}

Deep bidirectional LSTM, as illustrated in Fig. 4.

Long Short-Term Memory (LSTM)

\begin{align*}
    i_t &= \sigma(x_t W_t i + h_{t-1} W_t i + b_t) \\
    f_t &= \sigma(x_t W_t f + h_{t-1} W_t f + b_t) \\
    o_t &= \sigma(x_t W_t o + h_{t-1} W_t o + b_t) \\
    c_t &= f_t c_{t-1} + i_t \text{tanh}(x_t W_t c) \\
    h_t &= o_t \text{tanh}(c_t) \\
    y_t &= \sigma(h_t W_t y + b_t)
\end{align*}

Deep bidirectional LSTM, as illustrated in Fig. 4.

Long Short-Term Memory (LSTM)

\begin{align*}
    i_t &= \sigma(x_t W_t i + h_{t-1} W_t i + b_t) \\
    f_t &= \sigma(x_t W_t f + h_{t-1} W_t f + b_t) \\
    o_t &= \sigma(x_t W_t o + h_{t-1} W_t o + b_t) \\
    c_t &= f_t c_{t-1} + i_t \text{tanh}(x_t W_t c) \\
    h_t &= o_t \text{tanh}(c_t) \\
    y_t &= \sigma(h_t W_t y + b_t)
\end{align*}

Deep bidirectional LSTM, as illustrated in Fig. 4.
3. NETWORK TRAINING

Network training follows the standard approach used in hybrid systems [4]. Frame-level state targets are provided on the training set by a forced alignment given by a GMM-HMM system. The network is then trained to minimise the cross-entropy error of the targets using a softmax output layer with as many units as the total number of possible HMM states. At decoding time, the state probabilities yielded by the network are combined with a dictionary and language model to determine the most probable transcription. For a length \( T \) acoustic sequence \( x \), the network produces a length \( T \) output sequence \( y \), where each \( y_t \) defines a probability distribution over the \( K \) possible states: that is, \( y_k^t \) (the \( k \)th element of \( y_t \)) is the network’s estimate for the probability of observing state \( k \) at time \( t \) given \( x \). Given a length \( T \) state target sequence \( z \), the network is trained to minimise the negative log-probability of the target sequence given the input sequence:

\[
\log \Pr(z|x) = \sum_{t=1}^{T} \log y^z_t^t
\]

Which leads to the following error derivatives at the output layer

\[
\frac{\partial \log \Pr(z|x)}{\partial \hat{y}^k_t} = y^k_t^t
\]

where \( \hat{y}^t \) is the vector of output activations before they have been normalised with the softmax function. These derivatives are then fed back through the network using backpropagation through time to determine the weight gradient.

When training deep networks in hybrid systems with stochastic gradient descent it has been found advantageous to select minibatches of frames randomly from the whole training set, rather than using whole utterances as batches. This is impossible with RNN-HMM hybrids because the weight gradients are a function of the entire utterance.

Another difference is that hybrid deep networks are trained with an acoustic context window of frames to either side of the one being classified. This is not necessary for DBLSTM, since it is as able to store past and future context internally, and the data was therefore presented a single frame at a time.

For some of the experiments Gaussian noise was added to the network weights during training [15]. The noise was added once per training sequence, rather than at every timestep. Weight noise tends to ‘simplify’ neural networks, in the sense of reducing the amount of information required to transmit the parameters [16, 17], which improves generalisation.

4. TIMIT EXPERIMENTS

The first set of experiments were carried out on the TIMIT [18] speech corpus. Their purpose was to see how hybrid training for deep bidirectional LSTM compared with the end-to-end training methods described in [1]. To this end, we ensured that the data preparation, network architecture and training parameters were consistent with those in the previous work. To allow us to test for significance, we also carried out repeated runs of the previous experiments (which were only run once in the original paper). In addition, we ran hybrid experiments using a deep bidirectional RNN with \( \tanh \) hidden units instead of LSTM.

The standard 462 speaker set with all SA records removed was used for training, and a separate development set of 50 speakers was used for early stopping. Results are reported for the 24-speaker core test set. The audio data was preprocessed using a Fourier-transform-based filterbank with 40 coefficients (plus energy) distributed on a mel-scale, together with their first and second temporal derivatives. Each input layer consists of a combination of the 40 coefficients, the corresponding first and second temporal derivatives, and the energy.

Figure from (Graves et al., 2013)

- Figure: input/output layers not shown
- **Same general topology** as a Deep Bidirectional RNN, but with **LSTM units** in the hidden layers
- No additional **representational power** over DBRNN, but **easier to learn** in practice
How important is this particular architecture?

Jozefowicz et al. (2015) evaluated 10,000 different LSTM-like architectures and found several variants that worked just as well on several tasks.
RNN Training Tricks

• Deep Learning models tend to consist largely of matrix multiplications

• Training tricks:
  – mini-batching with masking
  – sorting into buckets of similar-length sequences, so that mini-batches have same length sentences
  – truncated BPTT, when sequences are too long, divide sequences into chunks and use the final vector of the previous chunk as the initial vector for the next chunk (but don’t backprop from next chunk to previous chunk)

<table>
<thead>
<tr>
<th></th>
<th>Metric</th>
<th>DyC++</th>
<th>DyPy</th>
<th>Chainer</th>
<th>DyC++ Seq</th>
<th>Theano</th>
<th>TF</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNNLM (MB=1)</td>
<td>words/sec</td>
<td>190</td>
<td>190</td>
<td>114</td>
<td>494</td>
<td>189</td>
<td>298</td>
</tr>
<tr>
<td>RNNLM (MB=4)</td>
<td>words/sec</td>
<td>830</td>
<td>825</td>
<td>295</td>
<td>1510</td>
<td>567</td>
<td>473</td>
</tr>
<tr>
<td>RNNLM (MB=16)</td>
<td>words/sec</td>
<td>1820</td>
<td>1880</td>
<td>794</td>
<td>2400</td>
<td>1100</td>
<td>606</td>
</tr>
<tr>
<td>RNNLM (MB=64)</td>
<td>words/sec</td>
<td>2440</td>
<td>2470</td>
<td>1340</td>
<td>2820</td>
<td>1260</td>
<td>636</td>
</tr>
</tbody>
</table>

Table from Neubig et al. (2017)
RNN Summary

• **RNNs**
  – Applicable to tasks such as **sequence labeling**, speech recognition, machine translation, etc.
  – Able to **learn context features** for time series data
  – Vanishing gradients are still a problem – but **LSTM units** can help

• **Other Resources**
  – Christopher Olah’s blog post on LSTMs
    [http://colah.github.io/posts/2015-08-Understanding-LSTMs/](http://colah.github.io/posts/2015-08-Understanding-LSTMs/)
MODULE-BASED AUTOMATIC DIFFERENTIATION
Backpropagation

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation
1. Write an **algorithm** for evaluating the function $y = f(x)$. The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the “**computation graph**”)
2. Visit each node in **topological order**.
   For variable $u_i$ with inputs $v_1,\ldots, v_N$
   a. Compute $u_i = g_i(v_1,\ldots, v_N)$
   b. Store the result at the node

Backward Computation (Version A)
1. **Initialize** $dy/dy = 1$.
2. Visit each node $v_j$ in **reverse topological order**.
   Let $u_1,\ldots, u_M$ denote all the nodes with $v_j$ as an input
   Assuming that $y = h(u) = h(u_1,\ldots, u_M)$
   and $u = g(v)$ or equivalently $u_i = g_i(v_1,\ldots, v_j,\ldots, v_N)$ for all $i$
   a. We already know $dy/du_i$ for all $i$
   b. Compute $dy/dv_j$ as below (Choice of algorithm ensures computing $(du_i/dv_j)$ is easy)
   \[
   \frac{dy}{dv_j} = \sum_{i=1}^{M} \frac{dy}{du_i} \frac{du_i}{dv_j}
   \]

Return partial derivatives $dy/du_i$ for all variables
Backpropagation

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

**Forward Computation**
1. Write an **algorithm** for evaluating the function $y = f(x)$. The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the “**computation graph**”)
2. Visit each node in **topological order**. For variable $u_i$ with inputs $v_1, \ldots, v_N$
   a. Compute $u_i = g_i(v_1, \ldots, v_N)$
   b. Store the result at the node

**Backward Computation (Version B)**
1. **Initialize** all partial derivatives $\frac{dy}{du_j}$ to 0 and $\frac{dy}{dy} = 1$
2. Visit each node in **reverse topological order**. For variable $u_i = g_i(v_1, \ldots, v_N)$
   a. We already know $\frac{dy}{du_i}$
   b. Increment $\frac{dy}{dv_j}$ by $(\frac{dy}{du_i})(\frac{du_i}{dv_j})$ (Choice of algorithm ensures computing $(\frac{du_i}{dv_j})$ is easy)

**Return** partial derivatives $\frac{dy}{du_i}$ for all variables
Why is the backpropagation algorithm efficient?

1. Reuses computation from the forward pass in the backward pass

2. Reuses partial derivatives throughout the backward pass (but only if the algorithm reuses shared computation in the forward pass)

(Key idea: partial derivatives in the backward pass should be thought of as variables stored for reuse)
A Recipe for Machine Learning

1. Given training data:
   \[ \{x_i, y_i\}_{i=1}^{N} \]

2. Choose each of these:
   - Decision function
   \[ \hat{y} = f_{\theta}(x_i) \]
   - Loss function
   \[ \ell(\hat{y}, y_i) \in \mathbb{R} \]

3. Define goal:
   \[ \text{Backpropagation} \text{ can compute this gradient!} \]
   And it’s a **special case of a more general algorithm** called reverse-mode automatic differentiation that can compute the gradient of any differentiable function efficiently!
Backpropagation: Abstract Picture

Forward
1. \(a = \alpha x\)
2. \(z = \sigma(a)\)
3. \(b = \beta z\)
4. \(\hat{y} = \text{softmax}(b)\)
5. \(J = -y^T \log \hat{y}\)

Backward
6. \(g_y = -y \div \hat{y}\)
7. \(g_b = g_y^T (\text{diag}(\hat{y}) - \hat{y}\hat{y}^T)\)
8. \(g_\beta = g_b^T z^T\)
9. \(g_z = \beta^T g_b\)
10. \(g_a = g_z \odot z \odot (1 - z)\)
11. \(g_\alpha = g_a x^T\)

\(J = \sum_{k=1}^{K} y_k^* \log(y_k)\)
\(y_k = \frac{\exp(b_k)}{\sum_{i=1}^{K} \exp(b_i)}\)
\(b_k = \sum_{j=0}^{D} \beta_{kj} z_j \quad \forall k\)
\(z_j = \sigma(a_j), \quad \forall j\)
\(a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \quad \forall j\)

Given \(x_i, \quad \forall i\)
Backpropagation: Procedural Method

### Drawbacks of Procedural Method

1. Hard to reuse / adapt for other models
2. (Possibly) harder to make individual steps more efficient
3. Hard to find source of error if finite-difference check reports an error (since it tells you only that there is an error somewhere in those 17 lines of code)

---

**Algorithm 1** Forward Computation

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>procedure</strong> NNFORWARD(Training example ((x, y)), Params (\alpha, \beta))</td>
</tr>
<tr>
<td>2</td>
<td>(a = \alpha x)</td>
</tr>
<tr>
<td>3</td>
<td>(z = \sigma(a))</td>
</tr>
<tr>
<td>4</td>
<td>(b = \beta z)</td>
</tr>
<tr>
<td>5</td>
<td>(\hat{y} = \text{softmax}(b))</td>
</tr>
<tr>
<td>6</td>
<td>(J = -y^T \log \hat{y})</td>
</tr>
<tr>
<td>7</td>
<td>(o = \text{object}(x, a, z, b, \hat{y}, J))</td>
</tr>
<tr>
<td>8</td>
<td><strong>return</strong> intermediate quantities (o)</td>
</tr>
</tbody>
</table>

**Algorithm 2** Backpropagation

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>procedure</strong> NNBACKWARD(Training example ((x, y)), Params (\alpha, \beta, ) Intermediates (o))</td>
</tr>
<tr>
<td>2</td>
<td>Place intermediate quantities (x, a, z, b, \hat{y}, J) in (o) in scope</td>
</tr>
<tr>
<td>3</td>
<td>(g_{\hat{y}} = -y \div \hat{y})</td>
</tr>
<tr>
<td>4</td>
<td>(g_b = g_{\hat{y}}^T (\text{diag}(\hat{y}) - \hat{y}\hat{y}^T))</td>
</tr>
<tr>
<td>5</td>
<td>(g_{\beta} = g_b^T z^T)</td>
</tr>
<tr>
<td>6</td>
<td>(g_z = \beta^T g_b)</td>
</tr>
<tr>
<td>7</td>
<td>(g_a = g_z \odot z \odot (1 - z))</td>
</tr>
<tr>
<td>8</td>
<td>(g_{\alpha} = g_a x^T)</td>
</tr>
<tr>
<td>9</td>
<td><strong>return</strong> parameter gradients (g_{\alpha}, g_{\beta})</td>
</tr>
</tbody>
</table>
Module-based AutoDiff

Module-based automatic differentiation (AD / Autodiff) is a technique that has long been used to develop libraries for deep learning

- **Dynamic neural network packages** allow a specification of the computation graph dynamically at runtime
  - PyTorch [http://pytorch.org](http://pytorch.org)
  - Torch [http://torch.ch](http://torch.ch)
  - DyNet [https://dynet.readthedocs.io](https://dynet.readthedocs.io)

- **Static neural network packages** require a static specification of a computation graph which is subsequently compiled into code
  - TensorFlow [https://www.tensorflow.org](https://www.tensorflow.org)
  - Aesara (and Theano) [https://aesara.readthedocs.io](https://aesara.readthedocs.io)
  - *(These libraries are also module-based, but herein by “module-based AD” we mean the dynamic approach)*
Module-based AutoDiff

- **Key Idea:**
  - componentize the computation of the neural-network into layers
  - each layer consolidates multiple real-valued nodes in the computation graph (a subset of them) into one vector-valued node (aka. a **module**)

- Each **module** is capable of two actions:
  1. Forward computation of output $b = [b_1, \ldots, b_B]$ given input $a = [a_1, \ldots, a_A]$ via some differentiable function $f$. That is $b = f(a)$.
  2. Backward computation of the gradient of the input $g_a = \nabla_a J = \left[ \frac{dJ}{da_1}, \ldots, \frac{dJ}{da_A} \right]$ given the gradient of output $g_b = \nabla_b J = \left[ \frac{dJ}{db_1}, \ldots, \frac{dJ}{db_B} \right]$, where $J$ is the final real-valued output of the entire computation graph. This is done via the chain rule $\frac{dJ}{da_i} = \sum_{j=1}^{B} \frac{dJ}{db_j} \frac{db_j}{da_i}$ for all $i \in \{1, \ldots, A\}$. 
Module-based AutoDiff

Dimensions: input $a \in \mathbb{R}^A$, output $b \in \mathbb{R}^B$, gradient of output $g_a \triangleq \nabla_a J \in \mathbb{R}^A$, and gradient of input $g_b \triangleq \nabla_b J \in \mathbb{R}^B$.

Sigmoid Module The sigmoid layer has only one input vector $a$. Below $\sigma$ is the sigmoid applied element-wise, and $\odot$ is element-wise multiplication s.t. $u \odot v = [u_1 v_1, \ldots, u_M v_M]$.

1: **procedure** SIGMOIDFORWARD($a$)
2: \hspace{1em} $b = \sigma(a)$
3: \hspace{1em} **return** $b$
4: **procedure** SIGMOIDBACKWARD($a$, $b$, $g_b$)
5: \hspace{1em} $g_a = g_b \odot b \odot (1 - b)$
6: \hspace{1em} **return** $g_a$

Softmax Module The softmax layer has only one input vector $a$. For any vector $v \in \mathbb{R}^D$, we have that $\text{diag}(v)$ returns a $D \times D$ diagonal matrix whose diagonal entries are $v_1, v_2, \ldots, v_D$ and whose non-diagonal entries are zero.

1: **procedure** SOFTMAXFORWARD($a$)
2: \hspace{1em} $b = \text{softmax}(a)$
3: \hspace{1em} **return** $b$
4: **procedure** SOFTMAXBACKWARD($a$, $b$, $g_b$)
5: \hspace{1em} $g_a = g_b^T (\text{diag}(b) - bb^T)$
6: \hspace{1em} **return** $g_a$

Linear Module The linear layer has two inputs: a vector $a$ and parameters $\omega \in \mathbb{R}^{B \times A}$. The output $b$ is not used by LINEARBACKWARD, but we pass it in for consistency of form.

1: **procedure** LINEARFORWARD($a$, $\omega$)
2: \hspace{1em} $b = \omega a$
3: \hspace{1em} **return** $b$
4: **procedure** LINEARBACKWARD($a$, $\omega$, $b$, $g_b$)
5: \hspace{1em} $g_\omega = g_b a^T$
6: \hspace{1em} $g_a = \omega^T g_b$
7: \hspace{1em} **return** $g_\omega, g_a$

Cross-Entropy Module The cross-entropy layer has two inputs: a gold one-hot vector $a$ and a predicted probability distribution $\hat{a}$. It’s output $b \in \mathbb{R}$ is a scalar. Below $\div$ is element-wise division. The output $b$ is not used by CROSSENTROPYBACKWARD, but we pass it in for consistency of form.

1: **procedure** CROSSENTROPYFORWARD($a$, $\hat{a}$)
2: \hspace{1em} $b = -a^T \log \hat{a}$
3: \hspace{1em} **return** $b$
4: **procedure** CROSSENTROPYBACKWARD($a$, $\hat{a}$, $b$, $g_b$)
5: \hspace{1em} $g_a = -g_b (a \div \hat{a})$
6: \hspace{1em} **return** $g_a$
Module-based AutoDiff

**Advantages of Module-based AutoDiff**

1. Easy to reuse / adapt for other models
2. Encapsulated layers are easier to optimize (e.g. implement in C++ or CUDA)
3. Easier to find bugs because we can run a finite-difference check on each layer separately
Module-based AutoDiff (OOP Version)

Object-Oriented Implementation:

– Let each module be an object
– Then allow the control flow dictate the creation of the computation graph
– No longer need to implement NNBackward(·), just follow the computation graph in reverse topological order
Module-based AutoDiff (OOP Version)

Object-Oriented Implementation:
– Let each module be an object
– Then allow the control flow dictate the creation of the computation graph
– No longer need to implement NNBackward(·), just follow the computation graph in reverse topological order

```python
class Sigmoid(Module):
    def forward(a):
        b = σ(a)
        return b
    def backward(a, b, gb):
        ga = gb * b * (1 - b)
        return ga

class Softmax(Module):
    def forward(a):
        b = softmax(a)
        return b
    def backward(a, b, gb):
        ga = gb.T * (diagonal(b) - b*b.T)
        return ga

class Linear(Module):
    def forward(a, ω):
        b = ω * a
        return b
    def backward(a, ω, b, gb):
        gω = gb * a.T
        ga = ω.T * gb
        return gω, ga

class CrossEntropy(Module):
    def forward(a, â):
        b = -a.T * log(â)
        return b
    def backward(a, â, b, gb):
        ga = -gb * (a / â)
        return ga
```
Module-based AutoDiff (OOP Version)

class NeuralNetwork(Module):
    
    method init():
        lin1_layer = Linear()
        sig_layer = Sigmoid()
        lin2_layer = Linear()
        soft_layer = Softmax()
        ce_layer = CrossEntropy()

    method forward(Tensor x, Tensor y, Tensor α, Tensor β):
        a = lin1_layer.apply_fwd(x, α)
        z = sig_layer.apply_fwd(a)
        b = lin1_layer.apply_fwd(z, β)
        ŷ = soft_layer.apply_fwd(b)
        J = ce_layer.apply_fwd(y, ŷ)
        return J.out_tensor

    method backward(Tensor x, Tensor y, Tensor α, Tensor β):
        tape_bwd()
        return lin1_layer.in_gradients[1], lin2_layer.in_gradients[1]
Module-based AutoDiff (OOP Version)

global tape = stack()

class Module:

    method init():
        out_tensor = null
        out_gradient = 1

    method apply_fwd(List in_modules):
        in_tensors = [x.out_tensor for x in in_modules]
        out_tensor = forward(in_tensors)
        tape.push(self)
        return self

    method apply_bwd():
        in_gradients = backward(in_tensors, out_tensor, out_gradient)
        for i in 1,...,len(in_modules):
            in_modules[i].out_gradient += in_gradients[i]
        return self

function tape_bwd():
    while len(tape) > 0
        m = tape.pop()
        m.apply_bwd()
PyTorch

• Q: Why don’t we call linear.forward() in PyTorch?
• A: This is just syntactic sugar. There’s a special method in Python __call__ that allows you to define what happens when you treat an object as if it were a function.

In other words, running the following:
   linear(x)
is equivalent to running:
   linear.__call__(x)
which in PyTorch is (nearly) the same as running:
   linear.forward(x)

This is because PyTorch defines every Module’s __call__ method to be something like this:
   def __call__(self):
       self.forward()
PyTorch

• Q: Why don’t we pass in the parameters to a PyTorch Module?

• A: This just makes your code cleaner.

In PyTorch, you store the parameters inside the Module and “mark” them as parameters that should contribute to the eventual gradient used by an optimizer.
Q&A