Gated Recurrent Models

Stephan Gouws & Richard Klein
Outline

Part 1: Intuition, Inference and Training

- Building intuitions: From Feedforward to Recurrent Models
- Inference in RNNs: Fprop
- Training in RNNs: Backpropagation-through-time (BPTT)

SHORT BREAK
Part 2: Gated models & Applications

- Long Short-term Memory (LSTMs)
- Gated Recurrent Units (GRUs)
- Applications:
  - Image captioning
  - Sequence classification (Practical 4: MNIST)
  - Language modeling
  - Sequence-labeling (lots of NLP tasks, e.g. POS tagging, NER, ...)
  - Sequence-to-sequence learning (Machine translation, Dialogue modeling, ...)

Outline
Recurrent Models

PART 1: Intuition, Inference and Training
Introduction
Introduction

28x28 x 2 pixels

Activation functions

Softmax

1 2 3 4

10 30 60 100 200 784
Introduction

Activation functions

28x28 x 3

pixels

softmax

1 2 ... 9

100 0 30 10

784 200
We need to be able to remember information from previous time steps
Recurrent neural networks: Intuition
Michel C. was born in Paris, France. He is married and has three children. He received a M.S. in neurosciences from the University Pierre & Marie Curie and the Ecole Normale Supérieure in 1987, and then spent most of his career in Switzerland, at the Ecole Polytechnique de Lausanne. He specialized in child and adolescent psychiatry and his first field of research was severe mood disorders in adolescent, topic of his PhD in neurosciences (2002). His mother tongue is ??? ????

Long-term dependencies: Why do they matter?

Michel C. was born in Paris, France. He is married and has three children. He received a M.S. in neurosciences from the University Pierre & Marie Curie and the Ecole Normale Supérieure in 1987, and then spent most of his career in Switzerland, at the Ecole Polytechnique de Lausanne. He specialized in child and adolescent psychiatry and his first field of research was severe mood disorders in adolescent, topic of his PhD in neurosciences (2002). His mother tongue is ??? ????

Long-term dependencies: Why do they matter?

Michel C. was born in Paris, France. He is married and has three children. He received a M.S. in neurosciences from the University Pierre & Marie Curie and the Ecole Normale Supérieure in 1987, and then spent most of his career in Switzerland, at the Ecole Polytechnique de Lausanne. He specialized in child and adolescent psychiatry and his first field of research was severe mood disorders in adolescent, topic of his PhD in neurosciences (2002). His mother tongue is ??? ????

Long-term dependencies: Why do they matter?

Michel C. was born in Paris, France. He is married and has three children. He received a M.S. in neurosciences from the University Pierre & Marie Curie and the Ecole Normale Supérieure in 1987, and then spent most of his career in Switzerland, at the Ecole Polytechnique de Lausanne. He specialized in child and adolescent psychiatry and his first field of research was severe mood disorders in adolescent, topic of his PhD in neurosciences (2002). His mother tongue is ??? ????

Long-term dependencies: Why do they matter?

Michel C. was born in Paris, France. He is married and has three children. He received a M.S. in neurosciences from the University Pierre & Marie Curie and the Ecole Normale Supérieure in 1987, and then spent most of his career in Switzerland, at the Ecole Polytechnique de Lausanne. He specialized in child and adolescent psychiatry and his first field of research was severe mood disorders in adolescent, topic of his PhD in neurosciences (2002). His mother tongue is ??? ????

Long-term dependencies: Why do they matter?

Michel C. was born in Paris, France. He is married and has three children. He received a M.S. in neurosciences from the University Pierre & Marie Curie and the Ecole Normale Supérieure in 1987, and then spent most of his career in Switzerland, at the Ecole Polytechnique de Lausanne. He specialized in child and adolescent psychiatry and his first field of research was severe mood disorders in adolescent, topic of his PhD in neurosciences (2002). His mother tongue is ??? ????

Long-term dependencies: Why do they matter?

Michel C. was born in Paris, France. He is married and has three children. He received a M.S. in neurosciences from the University Pierre & Marie Curie and the Ecole Normale Supérieure in 1987, and then spent most of his career in Switzerland, at the Ecole Polytechnique de Lausanne. He specialized in child and adolescent psychiatry and his first field of research was severe mood disorders in adolescent, topic of his PhD in neurosciences (2002). His mother tongue is ??? ????

Long-term dependencies: Why do they matter?

Michel C. was born in Paris, France. He is married and has three children. He received a M.S. in neurosciences from the University Pierre & Marie Curie and the Ecole Normale Supérieure in 1987, and then spent most of his career in Switzerland, at the Ecole Polytechnique de Lausanne. He specialized in child and adolescent psychiatry and his first field of research was severe mood disorders in adolescent, topic of his PhD in neurosciences (2002). His mother tongue is ??? ????

Long-term dependencies: Why do they matter?
Types of Sequence Models

FFNs

Image Captioning

MNIST predictor

Seq2Seq

Sequence labeling (e.g. NER)
Types of Sequence Models

- FFNs
- Image Captioning
- MNIST predictor
- Seq2Seq
- Sequence labeling (e.g., NER)
Types of Sequence Models

We’ll talk about this a little later. We’ll also implement this in today’s practical!
Types of Sequence Models

- **One to one**: FFNs, MNIST predictor
- **One to many**: Image Captioning
- **Many to one**: Seq2Seq
- **Many to many**:
Types of Sequence Models

- **one to one**
- **one to many**
- **many to one**
- **many to many**

**FFNs**

**Image Captioning**

**MNIST predictor**

**Seq2Seq**

**Sequence labeling** (e.g. NER)
Types of Sequence Models

- one to one: FFNs
- one to many: Image Captioning
- many to one: MNIST predictor
- many to many: Seq2Seq
- many to many: Sequence labeling (e.g. NER)
Classify following examples:

$$\{1, 9, 2, \ldots\}$$
FFNs vs RNNs

PREDICT: 1

Y: outputs

softmax

tanh

X: inputs

\{ \begin{bmatrix} \cdot \end{bmatrix}, \begin{bmatrix} \cdot \end{bmatrix}, \begin{bmatrix} \cdot \end{bmatrix}, \ldots \}
**FFNs vs RNNs**

- **X**: inputs
- **Y**: outputs

**PREDICT**: 9

- **softmax**
- **tanh**

- **H**: FFN

{1, 9, 4, ...}
FFNs vs RNNs

X: inputs

Y: outputs

PREDICT: 2

softmax

tanh

\{ 1, 0, \begin{array}{c} 1 \end{array}, \ldots \}
But what if these were not digits, but longer numbers?

Problem? Variable length inputs.
**RNNs vs FFNs**

- **RNN cell**
  - $Y_t$: outputs
  - $X_t$: inputs at step $t$
  - $H$

- **FFNs vs RNNs**

- **Softmax**
- **Tanh**

- $X$: inputs at step $t$

- $Y$: outputs

- $1?$
FFNs vs RNNs

X: inputs at step t
Y: outputs
H: internal state

RNN cell

softmax

H at t-1
X: inputs at step t

Yt

H

19?

"1"

tanh
Maintains a state (memory) that carries information between inputs!

\[ Y_t: \text{outputs} \]

\[ H: \text{internal state} \]

\[ X_t: \text{inputs at step } t \]

\[ H \text{ at } t-1 \]

\[ \text{softmax} \]

\[ \text{tanh} \]

FFNs vs RNNs
The RNN API

prev_state \rightarrow \text{recurrent_fn()} \rightarrow next_state

x \rightarrow \text{recurrent_fn()} \rightarrow outputs
The RNN Computation Graph

\[ f_\theta \]

\[ y_t \]

\[ x_t \]

"Feedback loop" / state / memory / stack (previous time-step)
“Unrolling” the RNN Computation Graph
Unrolling the RNN Computation Graph
Unrolling the RNN Computation Graph

“Unrolled” over $n$ time-steps.
Unrolling the RNN Computation Graph

"Unrolled" over $n$ time-steps.

NB: We reuse the same weights at every time-step!
Unrolling the RNN Computation Graph

We can therefore think of an RNN as a composition of identical feedforward neural networks (with replicated/tied weights), one for each moment or step in time.

NB: We reuse the same weights at every time-step!
The FFN API

class FeedForwardModel:

    # ...

def forward(self, x):
        # Compute activations on the hidden layer.
        hidden_layer = self.act_fn(np.dot(self.W_xh, x) + b)

        # Compute the (linear) output layer activations.
        y = np.dot(self.W_hy, hidden_layer)

        return y
class FeedForwardModel:

    # ...

    def forward(self, x):
        # Compute activations on the hidden layer.
        hidden_layer = self.act_fn(np.dot(self.W_xh, x) + b)

        # Compute the (linear) output layer activations.
        y = np.dot(self.W_hy, hidden_layer)

        return y
class FeedForwardModel():

    # ...

    def forward(self, x):
        # Compute activations on the hidden layer.
        hidden_layer = self.act_fn(np.dot(self.W_xh, x) + b)

        # Compute the (linear) output layer activations.
        y = np.dot(self.W_hy, hidden_layer)

        return y
The FFN API

class FeedForwardModel():

    # ...

    def forward(self, x):
        # Compute activations on the hidden layer.
        hidden_layer = self.act_fn(np.dot(self.W_xh, x) + b)

        # Compute the (linear) output layer activations.
        y = np.dot(self.W_hy, hidden_layer)

        return y
The FFN API

```python
class FeedForwardModel:
    # ...

def forward(self, x):
    # Compute activations on the hidden layer.
    hidden_layer = self.act_fn(np.dot(self.W_xh, x) + b)

    # Compute the (linear) output layer activations.
    y = np.dot(self.W_hy, hidden_layer)

    return y
```
class RecurrentModel():

    # ...

    def recurrent_fn(self, x, prev_state):
        # Compute the new state based on the previous state and current input.
        new_state = self.act_fn(np.dot(self.W_xh, x) + np.dot(self.W_hh, prev_state) + b)

        # Compute the output vector.
        y = np.dot(self.W_hy, new_state)

        return new_state, y
class RecurrentModel:
    # ...

    def recurrent_fn(self, x, prev_state):
        # Compute the new state based on the previous state and current input.
        new_state = self.act_fn(np.dot(self.W_xh, x) + np.dot(self.W_hh, prev_state) + b)

        # Compute the output vector.
        y = np.dot(self.W_hy, new_state)

        return new_state, y
The RNN API

class RecurrentModel():

    # ...

    def recurrent_fn(self, x, prev_state):
        # Compute the new state based on the previous state and current input.
        new_state = self.act_fn(np.dot(self.W_xh, x) + np.dot(self.W_hh, prev_state))

        # Compute the output vector.
        y = np.dot(self.W_hy, new_state)

        return new_state, y

\[
    h_t = f_\theta(W_{xh}x_t + W_{hh}h_{t-1})
\]

New state \hspace{1cm} \text{Recurrent function} \hspace{1cm} \text{Input at current time-step} \hspace{1cm} \text{Previous state}
class RecurrentModel():
    # ...

def recurrent_fn(self, x, prev_state):
    # Compute the new state based on the previous state and current input.
    new_state = self.act_fn(np.dot(self.W_xh, x) + np.dot(self.W_hh, prev_state))

    # Compute the output vector.
    y = np.dot(self.W_hy, new_state)
    return new_state, y
def forward(self, data_sequence, initial_state):
    state = initial_state
    all_states, all_ys = [state], []
    cache = []

    for x, y in data_sequence:
        new_state, y_pred = recurrent_fn(x, state)
        loss += cross_entropy(y_pred, y)

        cache.append((new_state, y_pred))
        state = new_state

    return loss, cache
The RNN API

def forward(self, data_sequence, initial_state):
    state = initial_state
    all_states, all_ys = [state], []
    cache = []

    for x, y in data_sequence:
        new_state, y_pred = recurrent_fn(x, state)
        loss += cross_entropy(y_pred, y)
        cache.append((new_state, y_pred))
        state = new_state

    return loss, cache
def forward(self, data_sequence, initial_state):
    state = initial_state
    all_states, all_ys = [state], []
    cache = []

    for x, y in data_sequence:
        new_state, y_pred = recurrent_fn(x, state)
        loss += cross_entropy(y_pred, y)

        cache.append((new_state, y_pred))
        state = new_state

    return loss, cache
Math: FFNs v RNNs

\[ h = f_{\theta}(W_{xh}x + b) \]

**NOTATION:** \( W_{xh} \) is a matrix that maps a vector \( x \) into a vector \( h \).
Math: FFNs v RNNs

\[ h = f_\theta(W_{xh}x + b) \]

**NOTATION:** \( W_{xh} \) is a matrix that maps a vector \( x \) into a vector \( h \).
Math: FFNs v RNNs

\[ h = f_{\theta}(W_{xh}x + b) \]

**NOTATION:** $W_{xh}$ is a matrix that maps a vector $x$ into a vector $h$. 

**Activation function**

**Input**
Math: FFNs v RNNs

Hidden layer

\[ h = f_\theta(W_{xh}x + b) \]

NOTATION: $W_{xh}$ is a matrix that maps a vector $x$ into a vector $h$. 
Math: FFNs v RNNs

Hidden layer

\[ h = f_\theta(W_{xh}x + b) \]

Activation function

Input

New state

\[ h_t = f_\theta(W_{xh}x_t) \]

Recurrent function

Input at current time-step

NOTATION: \( W_{xh} \) is a matrix that maps a vector \( x \) into a vector \( h \).
Math: FFNs v RNNs

Hidden layer

\[ h = f_\theta(W_{xh}x + b) \]

New state

\[ h_t = f_\theta(W_{xh}x_t + W_{hh}h_{t-1}) \]

NOTATION: \( W_{xh} \) is a matrix that maps a vector \( x \) into a vector \( h \).
Math: FFNs v RNNs

Hidden layer

\[ h = f_\theta(W_{xh}x + b) \]

New state

\[ h_t = f_\theta(W_{xh}x_t + W_{hh}h_{t-1}) \]

Input

\[ x_t \]

Activation function

Input at current time-step

Previous state

Recurrent function

"Recurrent" weights

NOTATION: \( W_{xh} \) is a matrix that maps a vector \( x \) into a vector \( h \).
Inference & Training

● How do we make predictions using RNNs?
  ○ Forward propagation: “Fprop”
  ○ Essentially a composition of functions: $a_2 = f_2(f_1(x))$.
  ○ We “unroll” the computational graph over time-steps.

● How do we train RNNs?
  ○ Backward propagation: “Backprop-through time”
  ○ We need to consider predictions over several time-steps!
  ○ Credit assignment over time.
  ○ We work backwards in time from the last state to the first.
Training: Ways to Train RNNs

- **Echo State Networks**: Initialize $W_{xh}$, $W_{hh}$, $W_{ho}$, carefully, then only train $W_{ho}$!
- **Backpropagation through time (BPTT)**: Propagate errors backwards through the unrolled graph.
- There are other options.
Training: ESNs

- Simple solution: don’t train the recurrent weights ($W_{hh}$ & $W_{xh}$)!
- Initialization very important.
- Super simple. However, with recent improvements in initialization etc, BPTT does better!
Inference & Training

● How do we make predictions using RNNs?
  ○ Forward propagation: “Fprop”
  ○ Essentially a composition of functions: \( a_2 = f_2(f_1(x)) \).
  ○ We “unroll” the computational graph over time-steps.

● How do we train RNNs?
  ○ Propagate errors backwards through unrolled graph: “Backprop-through time” (BPTT).
  ○ We need to consider predictions over several time-steps!
  ○ Credit assignment over time.
  ○ We work backwards in time from the last state to the first.
Training: BPTT Intuition

Backpropagation through time

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient
Training: **Truncated BPTT**

Run forward and backward through chunks of the sequence instead of whole sequence
Training: **Truncated BPTT**

Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps.
Training: **Truncated BPTT**
Unrolling the RNN Computation Graph

\[ h_0 \xrightarrow{f_\theta} h_1 \xrightarrow{f_\theta} h_2 \xrightarrow{f_\theta} \ldots \xrightarrow{f_\theta} h_t \]

\[ x_1 \xrightarrow{} y_1 \]
\[ x_2 \xrightarrow{} y_2 \]
\[ x_3 \xrightarrow{} y_t \]

“Unrolled” over \( n \) time-steps.
Unrolling the RNN Computation Graph

“Unrolled” over \( n \) time-steps.
Step 1: Compute all errors.

“Unrolled” over $n$ time-steps.
Unrolling the RNN Computation Graph

Unrolled over $n$ time-steps.

**Step 1**: Compute all errors.
**Step 2**: Pass error back for each time-step from $n$ back to 1.
Unrolling the RNN Computation Graph

Step 1: Compute all errors.
Step 2: Pass error back for each time-step from n back to 1.
Step 3: Update weights.

“Unrolled” over n time-steps.
Unrolling the RNN Computation Graph

\[ y_{t-2} \rightarrow h_{t-2} \xrightarrow{f_\theta} h_{t-1} \rightarrow y_{t-1} \]

\[ h_{t-1} \xrightarrow{f_\theta} h_t \rightarrow y_t \]

\[ x_{t-1} \rightarrow h_{t-2} \]

\[ x_t \rightarrow h_t \]
Unrolling the RNN Computation Graph

\[ y_{t-2} \]
\[ h_{t-2} \]
\[ \theta \]

\[ h_{t-1} \]
\[ f_\theta \]

\[ y_{t-1} \]

\[ h_{t} \]
\[ f_\theta \]

\[ y_{t} \]

\[ E_t \]
Unrolling the RNN Computation Graph

\[ h_{t-2} \xrightarrow{f_{\theta}} h_{t-1} \xrightarrow{f_{\theta}} h_t \]

\[ y_t \]

\[ E_t \]
Unrolling the RNN Computation Graph
Unrolling the RNN Computation Graph

\[ h_{t-2} \xrightarrow{f_\theta} h_{t-1} \xrightarrow{f_\theta} h_t \]

\[ y_t \]

\[ E_t \]

\[ \frac{\partial E_t}{\partial h_t} \]

\[ \frac{\partial h_t}{\partial \theta} \]
Unrolling the RNN Computation Graph

\[ h_{t-2} \xrightarrow{f_\theta} h_{t-1} \xrightarrow{f_\theta} h_t \xrightarrow{\frac{\partial h_t}{\partial h_{t-1}}} y_t \xrightarrow{\frac{\partial E_t}{\partial h_t}} E_t \]

\[ \frac{\partial h_t}{\partial \theta} \]
Unrolling the RNN Computation Graph
Unrolling the RNN Computation Graph

\[ h_{t-2} \rightarrow h_{t-1} \rightarrow h_t \rightarrow y_t \]

\[ \frac{\partial h_{t-1}}{\partial h_{t-2}} \rightarrow f_{\theta} \rightarrow \frac{\partial h_{t-1}}{\partial \theta} \rightarrow \frac{\partial h_{t-2}}{\partial \theta} \]

\[ \frac{\partial h_t}{\partial h_{t-1}} \rightarrow f_{\theta} \rightarrow \frac{\partial h_t}{\partial \theta} \]

\[ \frac{\partial E_t}{\partial h_t} \rightarrow E_t \]
Unrolling the RNN Computation Graph

\[ h_{t-2} \xrightarrow{f_\theta} h_{t-1} \xrightarrow{f_\theta} h_t \]  

\[ \frac{\partial h_{t-1}}{\partial h_{t-2}} \quad \frac{\partial h_t}{\partial h_{t-1}} \]

\[ \frac{\partial y_t}{\partial E_t} \]

\[ \frac{\partial E_t}{\partial \theta} = \sum_{t'=1}^{t} \frac{\partial E_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial h_{t'}}{\partial \theta} \]  

where

\[ \frac{\partial h_t}{\partial h_{t'}} = \prod_{k=t'+1}^{t} \frac{\partial h_k}{\partial h_{k-1}} \]
Unrolling the RNN Computation Graph

\[ \text{Total Error} = E_1 + E_2 + \ldots + E_t \]

\[ \text{Total gradient} = \text{sum of all } \frac{dE_t}{d\theta} \text{'s} \]

\[ \frac{\partial E_t}{\partial \theta} = \sum_{t'=1}^{t} \frac{\partial E_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t'}} \frac{\partial h_{t'}}{\partial \theta} \]

where

\[ \frac{\partial h_t}{\partial h_{t'}} = \prod_{k=t'+1}^{t} \frac{\partial h_k}{\partial h_{k-1}} \]
def bptt(model, X_train, y_train, initial_state):
    # Forward
    Loss, caches = forward(X_train, y_train, model, initial_state)
    avg_loss /= y_train.shape[0]
    # Backward
    dh_next = np.zeros((1, last_state.shape[0]))
    grads = {k: np.zeros_like(v) for k, v in model.items()}
    for t in reversed(range(len(X_train))):
        grad, dh_next = cell_fn_backward(ys[t], y_train[t], dh_next, caches[t])
        for k in grads.keys():
            grads[k] += grad[k]
    return grads, avg_loss

"Lego block"!
Training: **Truncated BPTT Code**

```python
def bptt(model, X_train, y_train, initial_state):
    # Forward
    Loss, caches = forward(X_train, y_train, model, initial_state)
    avg_loss /= y_train.shape[0]
    # Backward
    dh_next = np.zeros((1, last_state.shape[0]))
    grads = {k: np.zeros_like(v) for k, v in model.items()}

    for t in reversed(range(len(X_train))):
        grad, dh_next = cell_fn_backward(ys[t], y_train[t], dh_next, caches[t])
        for k in grads.keys():
            grads[k] += grad[k]

    return grads, avg_loss
```

"Lego block"!

Total gradient = Sum of these lego-gradients over time!
Vanilla RNN Gradient Flow

\[ h_t = \tanh(W_{hh} h_{t-1} + W_{hx} x_t) \]
\[ = \tanh \left( \begin{pmatrix} W_{hh} & W_{hx} \end{pmatrix} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \]
\[ = \tanh \left( W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right) \]
Vanilla RNN Gradient Flow

Backpropagation from $h_t$ to $h_{t-1}$ multiplies by $W$ (actually $W_{hh}^T$)

$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$

$= \tanh \left( (W_{hh} \quad W_{hx}) \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$

$= \tanh \left( W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$
Vanilla RNN Gradient Flow

Computing gradient of $h_0$ involves many factors of $W$ (and repeated tanh)
Vanilla RNN Gradient Flow

Computing gradient of $h_0$ involves many factors of $W$ (and repeated tanh)

Largest singular value $> 1$: Exploding gradients

Largest singular value $< 1$: Vanishing gradients
Vanilla RNN Gradient Flow

Computing gradient of $h_0$ involves many factors of $W$ (and repeated tanh)

Largest singular value > 1: Exploding gradients
Largest singular value < 1: Vanishing gradients

Gradient clipping: Scale gradient if its norm is too big

```python
grad_norm = np.sum(grad * grad)
if grad_norm > threshold:
    grad *= (threshold / grad_norm)
```
Vanilla RNN Gradient Flow

Computing gradient of $h_0$ involves many factors of $W$ (and repeated tanh)

Largest singular value $> 1$: Exploding gradients

Largest singular value $< 1$: Vanishing gradients

Change RNN architecture

Part 2!
Gated Recurrent Models

PART II: Gated Architectures & Applications
RECAP: The RNN API

\[ h_t = f_\theta(W_{xh}x_t + W_{hh}h_{t-1}) \]

prev_state \rightarrow \text{recurrent_fn()} \rightarrow next_state

x \rightarrow \text{recurrent_fn()} \rightarrow outputs

New state  Recurrent function  Input at current time-step  Previous state
The GATED RNN API

It is the same! Just a different way of computing the outputs.
Implementing a memory cell in a neural network

To preserve information for a long time in the activities of an RNN, we use a circuit that implements an analog memory cell.

- A linear unit that has a self-link with a weight of 1 will maintain its state.
- Information is stored in the cell by activating its write gate.
- Information is retrieved by activating the read gate.
- We can backpropagate through this circuit because logistics are have nice derivatives.
Propagating through a memory cell
Backpropagating through a memory cell?
LSTM = Long Short Term Memory

Concenate: \( X = X_t | H_{t-1} \)

Forget gate: \( f = \sigma(X.W_f + b_f) \)

Update gate: \( u = \sigma(X.W_u + b_u) \)

Result gate: \( r = \sigma(X.W_r + b_r) \)

Input: \( X' = \text{tanh}(X.W_c + b_c) \)

New C: \( C_t = f \times C_{t-1} + u \times X' \)

New H: \( H_t = r \times \text{tanh}(C_t) \)

Output: \( Y_t = \text{softmax}(H_t.W + b) \)
Element-wise operations

Neural net. layers
LSTM

\[ H_t = \sigma(\text{tanh}(X_t \circledast H_{t-1} + C_t \circledast C_{t-1})) \]

\[ Y_t = \sigma(\text{tanh}(X_t \circledast H_t)) \]

Element-wise operations

Neural net. layers
What to forget?

Element-wise operations

Neural net. layers
Actually forget

Element-wise operations
What to update?

Element-wise operations

Neural net. layers
LSTM

\[ X_t \]

\[ H_t \]

\[ Y_t \]

\[ C_t \]

Element-wise operations

Neural net. layers

Actually update
Element-wise operations

Neural net. layers

Updated!

\[
\begin{align*}
X_t & \quad H_{t-1} & \quad C_{t-1} \\
\sigma & \quad \sigma & \quad \tanh & \quad \sigma \\
\times & \quad \times & \quad \times & \quad \tanh & \quad \times \\
\& & & & & \quad \tanh \\
Y_t & \quad H_t & \quad C_t
\end{align*}
\]
LSTM

\[ \text{Element-wise operations} \]

\[ \text{tanh} \]

\[ \sigma \]

Neural net. layers

\[ \times \]

\[ \text{Result Gate!} \]

Calculate Result

\[ \text{Yt} \]
LSTM

\[ \text{X}_t \]

\[ \text{H}_{t-1} \]

\[ \text{C}_{t-1} \]

\[ \sigma \]

\[ \text{tanh} \]

\[ \times \]

\[ \sigma \]

\[ \times \]

\[ + \]

\[ \text{tanh} \]

\[ \text{H}_t \]

\[ \text{C}_t \]

\[ \text{Y}_t \]

Remember the result for next time step

Element-wise operations

Neural net. layers
LSTM = Long Short Term Memory

\[ X = X_t \oplus H_{t-1} \]

\[ f = \sigma(X.W_f + b_f) \]

\[ u = \sigma(X.W_u + b_u) \]

\[ r = \sigma(X.W_r + b_r) \]

\[ X' = \tanh(X.W_c + b_c) \]

\[ C_t = f * C_{t-1} + u * X' \]

\[ H_t = r * \tanh(C_t) \]

\[ Y_t = \text{softmax}(H_t.W + b) \]
LSTM = Long Short Term Memory

\[
X = X_t \oplus H_{t-1}
\]

\[
f = \sigma(X.W_f + b_f)
\]

\[
u = \sigma(X.W_u + b_u)
\]

\[
r = \sigma(X.W_r + b_r)
\]

\[
X' = \tanh(X.W_c + b_c)
\]

\[
C_t = f \ast C_{t-1} + u \ast X'
\]

\[
H_t = r \ast \tanh(C_t)
\]

\[
Y_t = \text{softmax}(H_t.W + b)
\]
Gru!
Gated Recurrent Units (GRUs)

GRU = Gated Recurrent Unit

2 gates instead of 3 => cheaper

\[ X = x_t \mid H_{t-1} \]

\[ z = \sigma(X.W_z + b_z) \]
\[ r = \sigma(X.W_r + b_r) \]

\[ X' = x_t \mid r \ast H_{t-1} \]

\[ X'' = \tanh(X'.W_c + b_c) \]

\[ H_t = (1-z) \ast H_{t-1} + z \ast X'' \]

\[ Y_t = \text{softmax}(H_t.W + b) \]
Gated Recurrent Units (GRUs)

GRU = Gated Recurrent Unit

2 gates instead of 3 => cheaper

\[ X = x_t \ | \ H_{t-1} \]

\[ z = \sigma(X.W_z + b_z) \]

\[ r = \sigma(X.W_r + b_r) \]

\[ X' = x_t \ | \ r * H_{t-1} \]

\[ X'' = \tanh(X'.W_c + b_c) \]

\[ H_t = (1-z) * H_{t-1} + z * X'' \]

\[ Y_t = \text{softmax}(H_t.W + b) \]
Long Short-term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]

Backpropagation from $c_t$ to $c_{t-1}$ only elementwise multiplication by $f$, no matrix multiply by $W$

\[
\begin{align*}
(i) &= \left( \begin{array}{c}
\sigma \\
\sigma \\
\sigma \\
\tanh
\end{array} \right) \\
(f) &= \left( \begin{array}{c}
\sigma \\
\sigma \\
\sigma \\
\tanh
\end{array} \right) W \left( \begin{array}{c}
h_{t-1} \\
x_t
\end{array} \right)
\end{align*}
\]

\[
\begin{align*}
c_t &= f \odot c_{t-1} + i \odot g \\
h_t &= o \odot \tanh(c_t)
\end{align*}
\]
Long Short-term Memory (LSTM): Gradient Flow

[Hochreiter et al., 1997]

Uninterrupted gradient flow!
Applications/Tasks

- Image captioning
- Sequence classification (Practical 4: MNIST)
- Language modeling
- Sequence-labeling (lots of NLP tasks, e.g. POS tagging, NER, ...)
- Sequence-to-sequence learning (Machine translation, Summarization, ...)
One-to-Many: Image captioning

**GOAL:** Given image, generate a sentence to describe its content.
GOAL: Given image, generate a sentence to describe its content.

- "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."
Many-to-one: Sequence Classifier (Prac 4)

**GOAL:** Given a sequence of inputs, predict the label for the whole sequence.

Examples:

- Given a sentence, say if it is {negative, neutral, positive}
- Given the words in an email, predict if it is a spam message.
- Given “pieces” of an image, predict what number is in the image.
Many-to-1: Polarity/Sentiment Classifier

We feed all the words into the model one at a time, and make one prediction at the end:

Cats are awesome
Many-to-1: Spam Classifier

We feed all the words into the model one at a time, and make one prediction at the end:

Viagra for cheap
Many-to-1: Image Classifier (Prac 4)

We chop up the image and feed all the pieces through the model, and then make one prediction at the end:
Next-token Prediction: Language modeling

1. Computing $p(\text{next word} \mid \text{previous words})$

2. $p(x_1, x_2, \ldots, x_m) = \prod_{i}^{m} p(x_i \mid x_1 \ldots x_{i-1})$
Next-token Prediction: Language modeling
Many-to-many: Sequence labeling

- Mapping each input $x_1, x_2, ..., x_n$ to its own label $y_1, y_2, ..., y_n$
- (Notice: **Same length** m; each input has an output.)
- A lot of NLP Tasks fall in this category, e.g.:
  - **Part-of-speech tagging**: map words to their parts-of-speech (noun, verb, etc).
  - **Named-entity Recognition**: identify mentions of people, places, etc in text
  - **Semantic Role Labeling**: find the main **actions**, and **who** performs them on **whom/what**
Many-to-many: Sequence labeling

- Part-of-speech tagging
Many-to-many: Sequence labeling

- Part-of-speech tagging
Heat water in a large vessel.
Many-to-many: Sequence-to-Sequence Modeling

- Mapping input sequences \( \{x_1, x_2, \ldots, x_m\} \) to output sequences \( \{y_1, y_2, \ldots, y_n\} \).
- Note: Arbitrary \( m \) and \( n \).
- Many applications, most notably Machine Translation.
- Also: POS tagging, Parsing, Summarization, Dialog
Many-to-many: Sequence-to-Sequence Modeling

the blue house <s> la maison bleu <eos>

Encoder RNN
Decoder RNN

MACHINE TRANSLATION
Many-to-many: Sequence-to-Sequence Modeling

**Prompt:** You can’t eat pasta every day. You can’t live only on Italian food.

**System:** I don’t eat Italian food, but I do eat a lot of Italian food.
Prompt: What do you have planned tonight?

Baseline: I have no idea what I want to do with my life. I have no idea what I want to (Bad, Mediocre, Bad)

Ours: I was planning on doing a few of these, but I’m not sure what to choose. I need to find a place to camp, and I don’t have a lot of time to make a plan. (Excellent, Good, Mediocre)
Prompt: Have you seen *Suicide Squad*?

Baseline: I have not, but I will check it out. (Good, Good, Good)

Ours: I saw it in a theater once and it was great. It was very... I don’t know, a little dark. I like the psychological effects and the way it portrays the characters in the movie. (Excellent, Excellent, Excellent)
Key take-aways

- RNNs have memory/state that evolve over time.
- We unroll the graph over time to do forward propagation.
- Backprop-through-time (BPTT):
  - Perform Chain Rule over the unrolled graph efficiently by saving and reusing previous computations.
  - $\frac{dE}{dW}$ is sum over all time-steps (b/c of tied weights)
- Vanishing/exploding gradients
- Gated architectures
  - State is *selectively* overwritten per time-step
  - Uninterrupted gradient flow through time: no vanishing/exploding gradients!
- Lots of cool applications!
Slide Credits

Thank-you to the following resources, from which some of these slides were drawn and adapted.

- [Stanford CS231n](http://cs231n.stanford.edu/)
- [TensorFlow without a PhD](https://www.tensorflow.org/)
The end.