Advanced Data Science

Dr. Kira Radinsky

Slides Adapted from Richard Socher
Overview

• Traditional language models
• RNNs
• RNN language models
• Important training problems and tricks
  • Vanishing and exploding gradient problems
• Bidirectional RNNs
• RNNs for other sequence tasks
Language Models

A language model computes a probability for a sequence of words:  \( P(w_1, \ldots, w_T) \)

- Useful for machine translation
  - Word ordering:  
    \[ p(\text{the cat is small}) > p(\text{small the is cat}) \]
  - Word choice:  
    \[ p(\text{walking home after school}) > p(\text{walking house after school}) \]
Traditional Language Models

• Probability is usually conditioned on window of n previous words

• An incorrect but necessary Markov assumption!

\[ P(w_1, \ldots, w_m) = \prod_{i=1}^{m} P(w_i \mid w_1, \ldots, w_{i-1}) \approx \prod_{i=1}^{m} P(w_i \mid w_{i-(n-1)}, \ldots, w_{i-1}) \]

• To estimate probabilities, compute for unigrams and bigrams (conditioning on one/two previous word(s):

\[ p(w_2 \mid w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)} \quad p(w_3 \mid w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2)} \]
Traditional Language Models

• Performance improves with keeping around higher n-grams counts and doing smoothing and so-called backoff (e.g. if 4-gram not found, try 3-gram, etc)

• There are A LOT of n-grams!
  • Gigantic RAM requirements!

• Recent state of the art: *Scalable Modified Kneser-Ney Language Model Estimation* by Heafield et al.:
  • “Using one machine with 140 GB RAM for 2.8 days, we built an unpruned model on 126 billion tokens”
Original neural language model

A Neural Probabilistic Language Model, Bengio et al. 2003

\[ \hat{y} = \text{softmax} \left( W^{(2)} f \left( W^{(1)} x + b^{(1)} \right) + W^{(3)} x + b^{(3)} \right) \]

Original equations:

\[ y = b + Wx + U \tanh(d + Hx) \]

\[ P(w_t|w_{t-1}, \ldots, w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}. \]

Problem: Fixed window of context for conditioning :(

Table look-up in \( C \)
Matrix \( C \)
Matrix \( C \)

\[ \text{shared parameters across words} \]

\[ \text{index for } w_{t+1} \]
\[ \text{index for } w_{t-2} \]
\[ \text{index for } w_{t-4} \]
Recurrent Neural Networks!

- RNNs tie the weights at each time step
- Condition the neural network on all previous words
- RAM requirement only scales with number of words
Recurrent Neural Network language model

Given list of word vectors: \(x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots, x_T\)

At a single time step:

\[
h_t = \sigma \left( W^{(hh)} h_{t-1} + W^{(hx)} x_t \right)
\]

\[
\hat{y}_t = \text{softmax} \left( W^{(S)} h_t \right)
\]

\[
\hat{P}(x_{t+1} = v_j \mid x_t, \ldots, x_1) = \hat{y}_{t,j}
\]
Recurrent Neural Network language model

• Main idea: we use the same set of $W$ weights at all time steps!

• Everything else is the same:

$$h_t = \sigma \left( W^{(hh)} h_{t-1} + W^{(hx)} x_t \right)$$

$$\hat{y}_t = \text{softmax} \left( W^{(S)} h_t \right)$$

$$\hat{P}(x_{t+1} = v_j | x_t, \ldots, x_1) = \hat{y}_{t,j}$$

$h_{t-1} \in \mathbb{R}^{D_h}$ Output of the non-linear function at the previous time-step, $t - 1$.

“The document context score so far”.

$x_{[t]}$ is the column vector of $L$ at index $[t]$ at time step $t$

$W^{(hh)} \in \mathbb{R}^{D_h \times D_h}$ $W^{(hx)} \in \mathbb{R}^{D_h \times d}$ $W^{(S)} \in \mathbb{R}^{|V| \times D_h}$

weights matrix used to condition the output of the previous time-step, $h_{(t-1)}$

weights matrix used to condition the input word vector, $x_t$

$|V|$ is the vocabulary.
Recurrent Neural Network language model

\( \hat{y} \in \mathbb{R}^{|V|} \) is a probability distribution over the vocabulary. \( y_t \) is the distribution at each timestamp \( t \). Essentially, \( y_t \) is the next predicted word given the document context score so far (i.e. \( h_{t-1} \)) and the last observed word vector \( x^{(t)} \).
RNN Example

\( s_t = \tanh(Ux_t + Ws_{t-1}) \)
\( o_t = \text{softmax}(Vs_t) \)

Vocabulary size \( C = 8000 \)
A hidden layer size \( H = 100 \)

\[ x_t \in \mathbb{R}^{8000} \]
\[ o_t \in \mathbb{R}^{8000} \]
\[ s_t \in \mathbb{R}^{100} \]
\[ U \in \mathbb{R}^{100 \times 8000} \]
\[ V \in \mathbb{R}^{8000 \times 100} \]
\[ W \in \mathbb{R}^{100 \times 100} \]

U, V and W are the parameters of our network that we want to learn from data. Thus, we need to learn a total of \( 2HC + H^2 \) parameters = 1,610,000.
Recurrent Neural Network language model

Same cross entropy loss function but predicting words instead of classes

\[ J^{(t)}(\theta) = - \sum_{j=1}^{\left| V \right|} y_{t,j} \log \hat{y}_{t,j} \]
Recurrent Neural Network language model

Evaluation could just be negative of average log probability over dataset of size (number of words) $T$:

$$J = -\frac{1}{T} \sum_{t=1}^{T} \sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$$

But more common: Perplexity: $2^J$

Lower is better!
Training RNNs is hard

- Multiply the same matrix at each time step during forward prop

\[
\begin{align*}
&h_{t-1} \\
&y_{t-1} \\
&x_{t-1} \\
\end{align*}
\quad W \quad
\begin{align*}
&h_t \\
&y_t \\
&x_t \\
\end{align*}
\quad W \quad
\begin{align*}
&h_{t+1} \\
&y_{t+1} \\
&x_{t+1} \\
\end{align*}
\]

- Ideally inputs from many time steps ago can modify output \( y \)
- Take \( \frac{\partial E_2}{\partial W} \) for an example RNN with 2 time steps! Insightful!
The vanishing/exploding gradient problem

RNN goal is to enable propagating context information through faraway time-steps.

Recurrent neural networks propagate the same W matrix from one time-step to the next.
The vanishing/exploding gradient problem

Will RNN succeed in the same way in the following sentences?

Sentence 1
"Jane walked into the room. John walked in too. Jane said hi to ___"

Sentence 2
"Jane walked into the room. John walked in too. It was late in the day, and everyone was walking home after a long day at work. Jane said hi to ___"

For long sentences, the probability that "John" would be recognized as the next word reduces with the size of the context.
The vanishing gradient problem - Details

• Similar but simpler RNN formulation:

\[ h_t = W f(h_{t-1}) + W^{(h_x)} x_t \]
\[ \hat{y}_t = W^{(S)} f(h_t) \]

• Total error is the sum of each error at time steps t

\[ \frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W} \]

• Hardcore chain rule application:

\[ \frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W} \]
The vanishing gradient problem - Details

• Similar to backprop but less efficient formulation

• Useful for analysis we’ll look at:

\[
\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}
\]

• Remember:

\[
h_t = Wf(h_{t-1}) + W^{(hx)}x_t
\]

• More chain rule, remember:

\[
\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}
\]

• Each partial is a Jacobian:

\[
h \in \mathbb{R}^{D_n} : \frac{\partial h_j}{\partial h_{j-1}} = \begin{bmatrix} \frac{\partial h_{j,1}}{\partial h_{j-1,1}} & \cdots & \frac{\partial h_{j,D_n}}{\partial h_{j-1,D_n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{j,D_n}}{\partial h_{j-1,1}} & \cdots & \frac{\partial h_{j,D_n}}{\partial h_{j-1,D_n}} \end{bmatrix}
\]
The vanishing gradient problem - Details

- From previous slide: \( \frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \) h_{t-1} h_t

- Remember: \( h_t = Wf(h_{t-1}) + W^{(hx)}x_t \)

- To compute Jacobian, derive each element of matrix: \( \frac{\partial h_{j,m}}{\partial h_{j-1,n}} \)

\[
\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W^T \text{diag}[f'(h_{j-1})]
\]

- Where: \( \text{diag}(z) = \begin{pmatrix}
z_1 & 0 \\
z_2 & 0 \\
\vdots & \ddots \\
0 & \cdots & 0 & z_{n-1} \\
& & & & z_n
\end{pmatrix} \)

Check at home that you understand the diag matrix formulation
The vanishing gradient problem - Details

- Analyzing the norms of the Jacobians, yields:

\[ \left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq \|W^T\| \|\text{diag}[f'(h_{j-1})]\| \leq \beta_W \beta_h \]

- Where we defined \( \beta \)'s as upper bounds of the norms.
- The gradient is a product of Jacobian matrices, each associated with a step in the forward computation.

\[ \left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\beta_W \beta_h)^{t-k} \]

- This can become very small or very large quickly [Bengio et al 1994], and the locality assumption of gradient descent breaks down. \( \rightarrow \) Vanishing or exploding gradient
Why is the vanishing gradient a problem?

- The error at a time step ideally can tell a previous time step from many steps away to change during backprop
IPython Notebook with vanishing gradient example

- Example of simple and clean NNet implementation
- Comparison of sigmoid and ReLu units
- A little bit of vanishing gradient
RNN Python Example

The model: \[ s_t = \tanh(Ux_t + Ws_{t-1}) \]
\[ o_t = \text{softmax}(Vs_t) \]

Our loss:
\[ E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t \]
\[ E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t) = -\sum_t y_t \log \hat{y}_t \]
Here's an actual training example from our text:

x:
SENTENCE_START what are n't you understanding about this ? !
[0, 51, 27, 16, 10, 856, 53, 25, 34, 69]

y:
what are n't you understanding about this ? ! SENTENCE_END
[51, 27, 16, 10, 856, 53, 25, 34, 69, 1]
**initialize the weights** randomly in the interval $[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}]$, $n$- is the number of incoming connections from the previous layer

```python
def __init__(self, word_dim, hidden_dim=100, bptt_truncate=4):
    # Assign instance variables
    self.word_dim = word_dim
    self.hidden_dim = hidden_dim
    self.bptt_truncate = bptt_truncate
    # Randomly initialize the network parameters
    self.U = np.random.uniform(-np.sqrt(1./word_dim), np.sqrt(1./word_dim), (hidden_dim, word_dim))
    self.V = np.random.uniform(-np.sqrt(1./hidden_dim), np.sqrt(1./hidden_dim), (word_dim, hidden_dim))
    self.W = np.random.uniform(-np.sqrt(1./hidden_dim), np.sqrt(1./hidden_dim), (hidden_dim, hidden_dim))
```

**forward propagation** (predicting word probabilities)

```python
def forward_propagation(self, x):
    # The total number of time steps
    T = len(x)
    # During forward propagation we save all hidden states in s because need them later.
    # We add one additional element for the initial hidden, which we set to 0
    s = np.zeros((T+1, self.hidden_dim))
    s[-1] = np.zeros(self.hidden_dim)
    # The outputs at each time step. Again, we save them for later.
    o = np.zeros((T, self.word_dim))
    # For each time step...
    for t in range(T):
        # Note that we are indexing U by x[t]. This is the same as multiplying U with a one-hot vector.
        s[t] = np.tanh(self.U[:, x[t]] + self.W.dot(s[t-1]))
        o[t] = softmax(self.V.dot(s[t]))
    return o, s
```

# Each o_t is a vector of probabilities representing the words in our vocabulary, but sometimes, for example when evaluating our model, all we want is the next word with the highest probability:

```python
def predict(self, x):
    # Perform forward propagation and return index of the highest score
    o, s = self.forward_propagation(x)
    return np.argmax(o, axis=1)
```
For each word in the sentence (in our case: 45), our model made 8000 predictions representing probabilities of the next word (currently random as all U,V,W are random)

```
np.random.seed(10)
model = RNNumpy(vocabulary_size)
o, s = model.forward_propagation(X_train[10])
print o.shape
print o
```

```
45, 8000
[[ 0.00012408  0.0001244  0.00012603 ...,  0.00012515  0.00012488
  0.00012508]
 [ 0.00012536  0.00012582  0.00012436 ...,  0.00012482  0.00012456
  0.00012451]
 [ 0.00012387  0.0001252  0.00012474 ...,  0.00012559  0.00012588
  0.00012551]
 ..., 
 [ 0.00012414  0.00012455  0.0001252 ...,  0.00012487  0.00012494
  0.0001263 ]
 [ 0.0001252  0.00012393  0.00012509 ...,  0.00012407  0.00012578
  0.00012502]
 [ 0.00012472  0.0001253  0.00012487 ...,  0.00012463  0.00012536
  0.00012665]]
```
RNN Python Example: Loss Calculation

```python
def calculate_total_loss(self, x, y):
    L = 0
    # For each sentence...
    for i in np.arange(len(y)):
        o, s = self.forward_propagation(x[i])
        # We only care about our prediction of the "correct" words
        correct_word_predictions = o[np.arange(len(y[i])), y[i]]
        # Add to the loss based on how off we were
        L += -1.0 * np.sum(np.log(correct_word_predictions))
    return L

def calculate_loss(self, x, y):
    # Divide the total loss by the number of training examples
    N = np.sum([len(y_i) for y_i in y])
    return self.calculate_total_loss(x, y) / N
```

For random init: we have $C$ words in our vocabulary, so each word should be (on average) predicted with probability $1/C$, which would yield a loss of:

$$ L = -\frac{1}{N} N \log \frac{1}{C} = \log C $$

# Limit to 1000 examples to save time
print "Expected Loss for random predictions: \%f\" % np.log(vocabulary_size)
print "Actual loss: \%f\" % model.calculate_loss(X_train[:1000], y_train[:1000])

Expected Loss for random predictions: 8.987197
Actual loss: 8.987440
RNN Python Example: Training the RNN with SGD and Backpropagation Through Time (BPTT)

The model: 
\[ s_t = \tanh(Ux_t + Ws_{t-1}) \]
\[ o_t = \text{softmax}(Vs_t) \]

Our loss: 
\[ E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t \]
\[ E(y, \hat{y}) = \sum_i E_i(y_i, \hat{y}_i) = -\sum_i y_i \log \hat{y}_i \]

\[ z_3 = V s_3 \]
\[ \frac{\partial E_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial z_3} \frac{\partial z_3}{\partial V} = (\hat{y}_3 - y_3) \otimes s_3 \]
\[ \frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W} \]
\[ \delta_2^{(3)} = \frac{\partial E_3}{\partial z_2} = \frac{\partial E_3}{\partial s_3} \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial z_2} \]

\[ z_2 = U x_2 + W s_1 \]
Training the RNN with SGD and Backpropagation Through Time (BPTT): Calculate the Gradient

Find the parameters $U,V,W$ and $\theta$ that minimize the total loss $L$ on the training data using SGD.

```python
def bptt(self, x, y):
    T = len(y)
    # Perform forward propagation
    o = self.forward_propagation(x)
    # We accumulate the gradients in these variables
    dLdU = np.zeros(self.U.shape)
    dLdV = np.zeros(self.V.shape)
    dLdW = np.zeros(self.W.shape)
    delta_o = 0
    delta_o[np.arange(len(y)), y] = 1.
    # For each output backwards...
    for t in np.arange(T-1:-1:-1):
        dLdV += np.outer(delta_o[t], s[t].T)
        # Initial delta calculation
        delta_t = self.V.T.dot(delta_o[t]) * (1 - (s[t] ** 2))
        # Backpropagation through time (for at most self.bptt_truncate steps)
        for bptt_step in np.arange(max(0, t-self.bptt_truncate), t+1)[::-1]:
            # print "Backpropagation step t=%d bptt_step=%d % (t, bptt_step)
            dLdW += np.outer(delta_t, s[bptt_step]]
            dLdU[:, x[bptt_step]] += delta_t
            # Update delta for next step
            delta_t = self.W.T.dot(delta_t) * (1 - s[bptt_step-1] ** 2)
    return [dLdU, dLdV, dLdW]
```
Training the RNN with SGD and Backpropagation Through Time (BPTT): Gradient Checking

```python
def gradient_check(self, x, y, h=0.001, error_threshold=0.01):
    # Calculate the gradients using backpropagation. We want to checker if these are correct.
    bptt_gradients = self.bptt(x, y)
    # List of all parameters we want to check.
    model_parameters = ["W", "V", "W\]
    # Gradient check for each parameter
    for pdx, pname in enumerate(model_parameters):
        # Get the actual parameter value from the model, e.g. model.W
        parameter = operator.attrgetter(pname)(self)
        print "Performing gradient check for parameter %s with size %d." % (pname, np.prod(parameter.shape))
        # Iterate over each element of the parameter matrix, e.g. (0,0), (0,1), ...
        it = np.nditer(parameter, flags=['multi_index'], op_flags=['readwrite'])
        while not it.finished:
            ix = it.multi_index
            # Save the original value so we can reset it later
            original_value = parameter[ix]
            # Estimate the gradient using (f(x+h) - f(x-h))/(2*h)
            parameter[ix] = original_value + h
            gradplus = self.calculate_total_loss(x, y)
            parameter[ix] = original_value - h
            gradminus = self.calculate_total_loss(x, y)
            estimated_gradient = (gradplus - gradminus)/(2*h)
            # Reset parameter to original value
            parameter[ix] = original_value
            # The gradient for this parameter calculated using backpropagation
            backprop_gradient = bptt_gradients[pdx][ix]
            # Calculate the relative error: (|x - y|/(|x| + |y|))
            relative_error = np.abs(backprop_gradient - estimated_gradient)/(np.abs(backprop_gradient) + np.abs(estimated_gradient))
            # If the error is too large fail the gradient check
            if relative_error > error_threshold:
                print "Gradient Check ERROR: parameter=%s ix=%s" % (pname, ix)
                print "ah Loss: %.2f" % gradplus
                print "h Loss: %.2f" % gradminus
                print "Estimated Gradient: %.2f" % estimated_gradient
                print "Backpropagation Gradient: %.2f" % backprop_gradient
                print "Relative Error: %.2f" % relative_error
                return
            if rel_error < h
            # If the error is too small fail the gradient check
            return
        print "Gradient check for parameter %s passed." % (pname)

RNNumpy.gradient_check = gradient_check

# To avoid performing millions of expensive calculations we use a smaller vocabulary size for checking.
grad_check_vocab_size = 10
np.random.seed(10)
model = RNNumpy(grad_check_vocab_size, 10, bptt_truncate=100)
model.gradient_check([10, 1, 2, 3], [1, 2, 3, 4])
```

\[
\frac{\partial L}{\partial \theta} \approx \lim_{h \to 0} \frac{J(\theta + h) - J(\theta - h)}{2h}
\]
Training the RNN with SGD and Backpropagation Through Time (BPTT): SGD

```python
# Performs one step of SGD.
def numpy_sgd_step(self, x, y, learning_rate):
    # Calculate the gradients
dU, dV, dW = self.bptt(x, y)
    # Change parameters according to gradients and learning rate
    self.U -= learning_rate * dU
    self.V -= learning_rate * dV
    self.W -= learning_rate * dW

RNNumpy.sgd_step = numpy_sgd_step

# Outer SGD Loop
# - model: The RNN model instance
# - X_train: The training data set
# - y_train: The training data labels
# - learning_rate: Initial learning rate for SGD
# - epoch: Number of times to iterate through the complete dataset
# - evaluate_loss_after: Evaluate the loss after this many epochs
def train_with_sgd(model, X_train, y_train, learning_rate=0.005, epoch=100, evaluate_loss_after=5):
    # We keep track of the losses so we can plot them later
    losses = []
    num_examples_seen = 0
    for epoch in range(epoch):
        # Optionally evaluate the loss
        if (epoch % evaluate_loss_after == 0):
            loss = model.calculate_loss(X_train, y_train)
            losses.append((num_examples_seen, loss))
            time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            print "%s: Loss after num_examples_seen=%d epoch=%d: %f" % (time, num_examples_seen, epoch, loss)
        # Adjust the learning rate if loss increases
        if len(losses) &gt; 1 and losses[-1][1] &gt; losses[-2][1]:
            learning_rate = learning_rate * 0.5
            print "Setting learning rate to %f" % learning_rate
            sys.stdout.flush()
        # For each training example...
        for i in range(len(y_train)):
            # One SGD step
            model.sgd_step(X_train[i], y_train[i], learning_rate)
            num_examples_seen += 1
```

Training the RNN with SGD and Backpropagation Through Time (BPTT): SGD

```python
np.random.seed(10)
# Train on a small subset of the data to see what happens
model = RNNumpy(vocabulary_size)
losses = train_with_sgd(model, X_train[:100], y_train[:100], epoch=10, evaluat
```

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Loss after num_examples_seen=</th>
<th>epoch=</th>
<th>Loss after num_examples_seen=</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-09-30</td>
<td>10:08:19</td>
<td>0</td>
<td>0</td>
<td>8.987425</td>
</tr>
<tr>
<td>2015-09-30</td>
<td>10:08:35</td>
<td>0</td>
<td>1</td>
<td>8.976270</td>
</tr>
<tr>
<td>2015-09-30</td>
<td>10:08:50</td>
<td>0</td>
<td>2</td>
<td>8.960212</td>
</tr>
<tr>
<td>2015-09-30</td>
<td>10:09:06</td>
<td>0</td>
<td>3</td>
<td>8.930430</td>
</tr>
<tr>
<td>2015-09-30</td>
<td>10:09:22</td>
<td>0</td>
<td>4</td>
<td>8.862264</td>
</tr>
<tr>
<td>2015-09-30</td>
<td>10:09:38</td>
<td>0</td>
<td>5</td>
<td>6.913570</td>
</tr>
<tr>
<td>2015-09-30</td>
<td>10:10:07</td>
<td>0</td>
<td>6</td>
<td>6.302493</td>
</tr>
<tr>
<td>2015-09-30</td>
<td>10:10:24</td>
<td>0</td>
<td>7</td>
<td>6.014995</td>
</tr>
<tr>
<td>2015-09-30</td>
<td>10:10:39</td>
<td>0</td>
<td>8</td>
<td>5.833877</td>
</tr>
<tr>
<td>2015-09-30</td>
<td>10:10:39</td>
<td>0</td>
<td>9</td>
<td>5.710718</td>
</tr>
</tbody>
</table>

Seems the loss is decreasing!
def generate_sentence(model):
    # We start the sentence with the start token
    new_sentence = [word_to_index[sentence_start_token]]
    # Repeat until we get an end token
    while not new_sentence[-1] == word_to_index[sentence_end_token]:
        next_word_probs = model.forward_propagation(new_sentence)
        sampled_word = word_to_index[unknown_token]
        # We don't want to sample unknown words
        while sampled_word == word_to_index[unknown_token]:
            samples = np.random.multinomial(1, next_word_probs[-1])
            sampled_word = np.argmax(samples)
        new_sentence.append(sampled_word)
    sentence_str = [index_to_word[x] for x in new_sentence[1:-1]]
    return ' '.join(sentence_str)

num_sentences = 10
senten_min_length = 7

for i in range(num_sentences):
    sent = []
    # We want long sentences, not sentences with one or two words
    while len(sent) < senten_min_length:
        sent = generate_sentence(model)
    print ' '.join(sent)
A few selected (censored) sentences with capitalization for “reddit comments 2015-08”

1. Anyway, to the city scene you’re an idiot teenager.
2. What ? ! ! ! ! ignore!
3. Screw fitness, you’re saying: https
4. Thanks for the advice to keep my thoughts around girls.
5. Yep, please disappear with the terrible generation.

What went right?
Successfully learn syntax – correctly places commas, ends sentence with punctuation, multiple exclamation marks etc.

What went wrong?
Sentences don’t make sense or have grammatical errors!

Why?
Our vanilla RNN can’t generate meaningful text because it’s unable to learn dependencies between words that are several steps apart
In [21]:
    plt.plot(np.array(relu_array[:6000]), color='blue')
    plt.plot(np.array(sigm_array[:6000]), color='green')
    plt.title('Sum of magnitudes of gradients -- hidden layer neurons')

Out[21]: <matplotlib.text.Text at 0x10a331310>
Trick for exploding gradient: clipping trick

• The solution first introduced by Mikolov is to clip gradients to a maximum value.

\[
\text{Algorithm 1 Pseudo-code for norm clipping the gradients whenever they explode}
\]

\[
\hat{g} \leftarrow \frac{\partial \mathcal{E}}{\partial \theta} \\
\text{if } \|\hat{g}\| \geq \text{threshold} \text{ then} \\
\hat{g} \leftarrow \frac{\text{threshold} \hat{g}}{\|\hat{g}\|} \\
\text{end if}
\]

• Makes a big difference in RNNs.
Gradient clipping intuition

- Error surface of a single unit RNN running through a small number of time-steps
- High curvature walls
- Solid lines: standard gradient descent trajectories
- Dashed lines: gradients rescaled to fixed size

Figure from paper: On the difficulty of training Recurrent Neural Networks, Pascanu et al. 2013
For vanishing gradients: Initialization + ReLus!

- Initialize $W^{(*)}$'s to identity matrix $I$ and $f(z) = \text{rect}(z) = \max(z, 0)$
- $\rightarrow$ Huge difference!

- Initialization idea first introduced in * Parsing with Compositional Vector Grammars*, Socher et al. 2013

- New experiments with recurrent neural nets a year ago (!) in *A Simple Way to Initialize Recurrent Networks of Rectified Linear Units*, Le et al. 2015
Perplexity Results

KN5 = Count-based language model with Kneser-Ney smoothing & 5-grams

Table from paper *Extensions of recurrent neural network language model* by Mikolov et al 2011
Problem: Softmax is huge and slow

Trick: Class-based word prediction

\[ p(w_t|\text{history}) = p(c_t|\text{history})p(w_t|c_t) \]

\[ = p(c_t|h_t)p(w_t|c_t) \]

The more classes, the better perplexity but also worse speed:

<table>
<thead>
<tr>
<th>Classes</th>
<th>RNN</th>
<th>RNN+KN5</th>
<th>Min/epoch</th>
<th>Sec/test</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>134</td>
<td>112</td>
<td>12.8</td>
<td>8.8</td>
</tr>
<tr>
<td>50</td>
<td>136</td>
<td>114</td>
<td>9.8</td>
<td>6.7</td>
</tr>
<tr>
<td>100</td>
<td>136</td>
<td>114</td>
<td>9.1</td>
<td>5.6</td>
</tr>
<tr>
<td>200</td>
<td>136</td>
<td>113</td>
<td>9.5</td>
<td>6.0</td>
</tr>
<tr>
<td>400</td>
<td>134</td>
<td>112</td>
<td>10.9</td>
<td>8.1</td>
</tr>
<tr>
<td>1000</td>
<td>131</td>
<td>111</td>
<td>16.1</td>
<td>15.7</td>
</tr>
<tr>
<td>2000</td>
<td>128</td>
<td>109</td>
<td>25.3</td>
<td>28.7</td>
</tr>
<tr>
<td>4000</td>
<td>127</td>
<td>108</td>
<td>44.4</td>
<td>57.8</td>
</tr>
<tr>
<td>6000</td>
<td>127</td>
<td>109</td>
<td>70</td>
<td>96.5</td>
</tr>
<tr>
<td>8000</td>
<td>124</td>
<td>107</td>
<td>107</td>
<td>148</td>
</tr>
<tr>
<td>Full</td>
<td>123</td>
<td>106</td>
<td>154</td>
<td>212</td>
</tr>
</tbody>
</table>
Sequence modeling for other tasks

- Classify
  - NER
  - the sentiment of each word in its context
  - opinionated expressions

- Example application and slides from paper *Opinion Mining with Deep Recurrent Nets* by Irsoy and Cardie 2014
Opinion Mining with Deep Recurrent Nets

Goal: Classify each word as

*direct subjective expressions* (DSEs) and *
*expressive subjective expressions* (ESEs).

DSE: Explicit mentions of private states or speech events expressing private states

ESE: Expressions that indicate sentiment, emotion, etc. without explicitly conveying them.
Example Annotation

In BIO notation (tags either begin-of-entity (B\_X) or continuation-of-entity (I\_X)):
The committee, [as usual]_{ESE}, [has refused to make any statements]_{DSE}.

The committee, as usual, has refused to make any statements.

O O O B\_ESE I\_ESE O B\_DSE

refused to make any statements.

I\_DSE I\_DSE I\_DSE I\_DSE I\_DSE O
Approach: Recurrent Neural Network

- Notation from paper (so you get used to different ones)

  \[ h_t = f(Wx_t + Vh_{t-1} + b) \]
  \[ y_t = g(Uh_t + c) \]

- \( x \) represents a token (word) as a vector.

- \( y \) represents the output label (B, I or O) – \( g = \text{softmax} \) !

- \( h \) is the memory, computed from the past memory and current word. It summarizes the sentence up to that time.
Bidirectional RNNs

Problem: For classification you want to incorporate information from words **both preceding and following**

\[ h_t = f(W_x + V_{h_{t-1}} + b) \]

\[ \tilde{h}_t = f(W_x + \tilde{V}_{h_{t+1}} + \tilde{b}) \]

\[ y_t = g(U[h_t;\tilde{h}_t] + c) \]

\[ h = [\tilde{h}_t;\tilde{h}_t] \] now represents (summarizes) the past and future around a single token.
Deep Bidirectional RNNs

Each memory layer passes an intermediate sequential representation to the next.

\[
\begin{align*}
\overrightarrow{h}_t^{(i)} &= f(\overrightarrow{W}^{(i)} \overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{V}^{(i)} \overrightarrow{h}_{t+1}^{(i)} + \overrightarrow{b}^{(i)}) \\
\overleftarrow{h}_t^{(i)} &= f(\overleftarrow{W}^{(i)} \overleftarrow{h}_{t}^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t}^{(i)} + \overleftarrow{b}^{(i)}) \\
y_t &= g(U[\overrightarrow{h}_t^{(L)};\overleftarrow{h}_t^{(L)}] + c)
\end{align*}
\]
Data

- **MPQA 1.2 corpus** (Wiebe et al., 2005)
- consists of 535 news articles (11,111 sentences)
- manually labeled at the phrase level (with DSE and ESEs)
- **Evaluation:** F1

\[
\text{precision} = \frac{tp}{tp + fp} \\
\text{recall} = \frac{tp}{tp + fn} \\
F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}
\]
Notices that after several layers the results start to get worse (overfit?)
Summary

• Recurrent Neural Networks are powerful
• Next up: Fancy Recursive Neural Networks! (advanced stuff)
Advanced RNN (GRU and LSTM) for Machine Translation
Overview

• Machine translation

• RNN Models tackling MT:
  • Gated Recurrent Units by Cho et al. (2014)
  • Long-Short-Term-Memories by Hochreiter and Schmidhuber (1997)
Machine Translation

- Methods are statistical
- Use parallel corpora
  - European Parliament
- First parallel corpus:
  - Rosetta Stone →
- Traditional systems are very complex
Current statistical machine translation systems

- Source language $f$, e.g. French
- Target language $e$, e.g. English
- Probabilistic formulation (using Bayes rule)

$$\hat{e} = \arg\max_e p(e \mid f) = \arg\max_e p(f \mid e)p(e)$$

- Translation model $p(f \mid e)$ trained on parallel corpus
- Language model $p(e)$ trained on English only corpus (lots, free!)

Diagram:
- French $\rightarrow$ Translation Model $p(f \mid e)$ $\rightarrow$ Pieces of English $\rightarrow$ Language Model $p(e)$
- Decoder $\Rightarrow$ argmax $p(f \mid e)p(e)$ $\Rightarrow$ Proper English
Step 1: Alignment

Goal: know which word or phrases in source language would translate to what words or phrases in target language? → Hard already!

Alignment examples from Chris Manning/CS224n
Step 1: Alignment

“zero fertility” word not translated

And the program has been implemented

one-to-many alignment
Step 1: Alignment

Really hard :/

The \_\_\_\_ Le balance \_\_\_\_ reste
was the \_\_\_\_ appartenait
territory of \_\_\_\_ aux
the \_\_\_\_ autochtones
aboriginal \_\_\_\_ people

\[ \text{many-to-one alignments} \]
Step 1: Alignment

The poor don’t have any money

Les pauvres sont démunis
Step 1: Alignment

- We could spend an entire lecture on alignment models
- Not only single words but could use phrases, syntax
- Then consider reordering of translated phrases

Example from Philipp Koehn
After many steps

Each phrase in source language has many possible translations resulting in large search space:

**Translation Options**

<table>
<thead>
<tr>
<th>er</th>
<th>geht</th>
<th>ja</th>
<th>nicht</th>
<th>nach</th>
<th>hause</th>
</tr>
</thead>
<tbody>
<tr>
<td>he</td>
<td>is</td>
<td>yes</td>
<td>not</td>
<td>after</td>
<td>house</td>
</tr>
<tr>
<td>it</td>
<td>are</td>
<td>is</td>
<td>do not</td>
<td>to</td>
<td>home</td>
</tr>
<tr>
<td>it</td>
<td>goes</td>
<td>, of course</td>
<td>does not</td>
<td>according to</td>
<td>chamber</td>
</tr>
<tr>
<td>he</td>
<td>go</td>
<td>,</td>
<td>is not</td>
<td>in</td>
<td>at home</td>
</tr>
<tr>
<td>it is</td>
<td>he will be</td>
<td>,</td>
<td>not</td>
<td>under house</td>
<td></td>
</tr>
<tr>
<td>it goes</td>
<td>It goes</td>
<td>,</td>
<td>not</td>
<td>return home</td>
<td></td>
</tr>
<tr>
<td>he goes</td>
<td>he goes</td>
<td>,</td>
<td>not</td>
<td>do not</td>
<td></td>
</tr>
<tr>
<td>is</td>
<td>are</td>
<td>is</td>
<td>not</td>
<td>to</td>
<td></td>
</tr>
<tr>
<td>is after all</td>
<td>not after</td>
<td>,</td>
<td>not</td>
<td>following</td>
<td></td>
</tr>
<tr>
<td>does</td>
<td>does</td>
<td>does not</td>
<td>not to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>not</td>
<td>is not</td>
<td>not after</td>
<td></td>
<td></td>
</tr>
<tr>
<td>are not</td>
<td>are not</td>
<td>,</td>
<td>not to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is not</td>
<td>is not</td>
<td>,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is not</td>
<td>is not</td>
<td>,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>
Decode: Search for best of many hypotheses

Hard search problem that also includes language model
Traditional MT

- Skipped hundreds of important details
- **A lot** of human feature engineering
- Very complex systems
- Many different, independent machine learning problems
Deep learning to the rescue! … ?

Maybe, we could translate directly with an RNN?

Decoder:

Encoder

This needs to capture the entire phrase!
MT with RNNs – Simplest Model

Encoder: \[ h_t = \phi(h_{t-1}, x_t) = f\left( W^{(hh)}h_{t-1} + W^{(hx)}x_t \right) \]

Decoder: \[ h_t = \phi(h_{t-1}) = f\left( W^{(hh)}h_{t-1} \right) \]
[\[ y_t = \text{softmax}(W^{(S)}h_t) \]

Minimize cross entropy error for all target words conditioned on source words

\[ \max_{\theta} \frac{1}{N} \sum_{n=1}^{N} \log p_{\theta}(y^{(n)}|x^{(n)}) \]

It’s not quite that simple ;)}
RNN Translation Model Extensions

1. Train different RNN weights for encoding and decoding

This means the $\phi()$ functions in

$$h_t = \phi(h_{t-1}, x_t) = f(W^{hh}h_{t-1} + W^{hx}x_t)$$

would have different $W^{hh}$ matrices in the two models
RNN Translation Model Extensions

Notation: Each input of $\phi$ has its own linear transformation matrix. Simple: $h_t = \phi(h_{t-1}) = f\left(W^{(hh)}h_{t-1}\right)$

2. Compute every hidden state in decoder from

- Previous hidden state (standard)
- Last hidden vector of encoder $c=h_T$
- Previous predicted output word $y_{t-1}$

$$h_{D,t} = \phi_D(h_{t-1}, c, y_{t-1})$$

Language model with three inputs to each decoder neuron: $(h_{t-1}, c, y_{t-1})$

Cho et al. 2014
Different picture, same idea

$e = \text{(Economic, growth, has, slowed, down, in, recent, years, .)}$

$\mathbf{f} = \text{(La, croissance, économique, s’est, ralentie, ces, dernières, années, .)}$

Kyunghyun Cho et al. 2014
RNN Translation Model Extensions

3. Train stacked/deep RNNs with multiple layers

4. Potentially train bidirectional encoder

5. Train input sequence in reverse order for simple optimization problem: Instead of $A \, B \, C \rightarrow X \, Y$, train with $C \, B \, A \rightarrow X \, Y$
6. Main Improvement: Better Units

- More complex hidden unit computation in recurrence!
- Gated Recurrent Units (GRU)
  - introduced by Cho et al. 2014 (see reading list)
- Main ideas:
  - keep around memories to capture long distance dependencies
  - allow error messages to flow at different strengths depending on the inputs
GRUs

- Standard RNN computes hidden layer at next time step directly:
  \[ h_t = f \left( W^{(hh)} h_{t-1} + W^{(hx)} x_t \right) \]

- GRU first computes an update gate (another layer) based on current input word vector and hidden state
  \[ z_t = \sigma \left( W^{(z)} x_t + U^{(z)} h_{t-1} \right) \]

- Compute reset gate similarly but with different weights
  \[ r_t = \sigma \left( W^{(r)} x_t + U^{(r)} h_{t-1} \right) \]
GRUs

• Update gate
  \[ z_t = \sigma \left( W^{(z)} x_t + U^{(z)} h_{t-1} \right) \]

• Reset gate
  \[ r_t = \sigma \left( W^{(r)} x_t + U^{(r)} h_{t-1} \right) \]

• New memory content:
  \[ \tilde{h}_t = \tanh \left( W x_t + r_t \circ U h_{t-1} \right) \]

• Final memory at time step combines current and previous time steps:
  \[ h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t \]
GRUs

\[ z_t = \sigma \left( W^{(z)} x_t + U^{(z)} h_{t-1} \right) \]
\[ r_t = \sigma \left( W^{(r)} x_t + U^{(r)} h_{t-1} \right) \]
\[ \tilde{h}_t = \tanh \left( W x_t + r_t \circ U h_{t-1} \right) \]
\[ h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t \]

Intuitively, the \textbf{update gate} defines how much of the previous memory to keep around.
GRUs

\[ z_t = \sigma \left( W^{(z)} x_t + U^{(z)} h_{t-1} \right) \]
\[ r_t = \sigma \left( W^{(r)} x_t + U^{(r)} h_{t-1} \right) \]
\[ \tilde{h}_t = \tanh \left( W x_t + r_t \circ U h_{t-1} \right) \]
\[ h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t \]

Intuitively, the reset gate determines how to combine the new input with the previous memory.

If we set the reset to all 1’s and update gate to all 0’s we again arrive at our plain RNN model.
GRUs

• **Update gate**
  \[ z_t = \sigma \left( W^{(z)} x_t + U^{(z)} h_{t-1} \right) \]

• **Reset gate**
  \[ r_t = \sigma \left( W^{(r)} x_t + U^{(r)} h_{t-1} \right) \]

• **New memory content:**
  \[ \tilde{h}_t = \tanh \left( W x_t + r_t \odot U h_{t-1} \right) \]
  If reset gate unit is \( \sim 0 \), then this ignores previous memory and only stores the new word information
  
  if it the i-th element of \( r_t \) is 0 we only take the current word into account

• **Final memory at time step combines current and previous time steps:**
  \[ h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \]
  if it the i-th element of \( z_t \) is 1 we copy the previous state and ignore the current one (including the current word). Otherwise, we can take only the current word (based on previous reset gate) or with is connection to previous words
Attempt at a clean illustration

\[
\begin{align*}
    z_t &= \sigma \left( W^{(z)} x_t + U^{(z)} h_{t-1} \right) \\
    r_t &= \sigma \left( W^{(r)} x_t + U^{(r)} h_{t-1} \right) \\
    \tilde{h}_t &= \tanh \left( W x_t + r_t \circ U h_{t-1} \right) \\
    h_t &= z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t
\end{align*}
\]

Has to be sigmoid to illustrate the on/off switch better.
GRU intuition

- If reset is close to 0, ignore previous hidden state → Allows model to drop information that is irrelevant in the future
  
  \[
  z_t = \sigma \left( W^{(z)} x_t + U^{(z)} h_{t-1} \right)
  \]
  \[
  r_t = \sigma \left( W^{(r)} x_t + U^{(r)} h_{t-1} \right)
  \]
  \[
  \tilde{h}_t = \tanh \left( W x_t + r_t \circ U h_{t-1} \right)
  \]
  \[
  h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t
  \]

- Update gate \( z \) controls how much of past state should matter now.
  
  - If \( z \) close to 1, then we can copy information in that unit through many time steps! Less vanishing gradient!

- Units with short-term dependencies often have reset gates very active
GRU intuition

- Units with long term dependencies have active update gates $z$

- Illustration: 

- Derivative of $\frac{\partial}{\partial x_1} x_1 x_2$ ? $\rightarrow$ rest is same chain rule, but implement with **modularization** or automatic differentiation (e.g. theano)
GRU layer is just another way of computing the hidden state. So all we really need to do is change the hidden state computation in our forward propagation function.

```python
def forward_prop_step(x_t, s_t1_prev):
    # This is how we calculated the hidden state in a simple RNN. No longer!
    # s_t = T.tanh(U[:, x_t] + W.dot(s_t1_prev))

    # Get the word vector
    x_e = E[:, x_t]

    # GRU Layer
    z_t1 = T.nnet.hard_sigmoid(U[0].dot(x_e) + W[0].dot(s_t1_prev) + b[0])
    r_t1 = T.nnet.hard_sigmoid(U[1].dot(x_e) + W[1].dot(s_t1_prev) + b[1])
    c_t1 = T.tanh(U[2].dot(x_e) + W[2].dot(s_t1_prev * r_t1) + b[2])
    s_t1 = (T.ones_like(z_t1) - z_t1) * c_t1 + z_t1 * s_t1_prev

    # Final output calculation
    # Theano's softmax returns a matrix with one row, we only need the row
    o_t = T.nnet.softmax(V.dot(s_t1) + c)[0]

    return [o_t, s_t1]
```

In our implementation we also added bias units. It’s quite typical that these are not shown in the equations.
I also added a word embedding layer $E$. 
GRU Python Implementation: Gradients

We could derive the gradients for E,W,U,b and \( b \) by hand using the chain rule, just like we did before. But in practice most people use libraries like Theano that support auto-differenation of expressions.

```python
# Gradients using Theano
dE = T.grad(cost, E)
dU = T.grad(cost, U)
dW = T.grad(cost, W)
$db = T.grad(cost, b)$
dV = T.grad(cost, V)
dc = T.grad(cost, c)
```
Adding a second GRU layer

# GRU Layer 1
z_t1 = T.nnet.hard_sigmoid(U[0].dot(x_e) + W[0].dot(s_t1_prev) + b[0])
r_t1 = T.nnet.hard_sigmoid(U[1].dot(x_e) + W[1].dot(s_t1_prev) + b[1])
c_t1 = T.tanh(U[2].dot(x_e) + W[2].dot(s_t1_prev * r_t1) + b[2])
s_t1 = (T.ones_like(z_t1) - z_t1) * c_t1 + z_t1 * s_t1_prev

# GRU Layer 2
z_t2 = T.nnet.hard_sigmoid(U[3].dot(s_t1) + W[3].dot(s_t2_prev) + b[3])
r_t2 = T.nnet.hard_sigmoid(U[4].dot(s_t1) + W[4].dot(s_t2_prev) + b[4])
c_t2 = T.tanh(U[5].dot(s_t1) + W[5].dot(s_t2_prev * r_t2) + b[5])
s_t2 = (T.ones_like(z_t2) - z_t2) * c_t2 + z_t2 * s_t2_prev
Results

Here are a few good examples of the network output (capitalization added by me).

• I am a bot, and this action was performed automatically.
• I enforce myself ridiculously well enough to just youtube.
• I’ve got a good rhythm going!
• There is no problem here, but at least still wave!
• It depends on how plausible my judgement is.
• (with the constitution which makes it impossible)

Our network was able to **Semantic dependencies**! For example, bot and automatically are clearly related, as are the opening and closing brackets.
Long-short-term-memories (LSTMs)

• We can make the units even more complex

• Allow each time step to modify
  • Input gate (high if current cell matters)
    \[ i_t = \sigma \left( W^{(i)} x_t + U^{(i)} h_{t-1} \right) \]
  • Forget (gate 0, forget past)
    \[ f_t = \sigma \left( W^{(f)} x_t + U^{(f)} h_{t-1} \right) \]
  • Output (how much cell is exposed)
    \[ o_t = \sigma \left( W^{(o)} x_t + U^{(o)} h_{t-1} \right) \]
  • New memory cell
    \[ \tilde{c}_t = \tanh \left( W^{(c)} x_t + U^{(c)} h_{t-1} \right) \]
  • Final memory cell:
    \[ c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \]

• Final hidden state:
  \[ h_t = o_t \circ \tanh(c_t) \]

• Many variation: LSTM: A Search Space Odyssey
Long-short-term-memories (LSTMs)

\[
\begin{align*}
i_t &= \sigma \left( W^{(i)} x_t + U^{(i)} h_{t-1} \right) \\
f_t &= \sigma \left( W^{(f)} x_t + U^{(f)} h_{t-1} \right) \\
o_t &= \sigma \left( W^{(o)} x_t + U^{(o)} h_{t-1} \right) \\
\tilde{c}_t &= \tanh \left( W^{(c)} x_t + U^{(c)} h_{t-1} \right) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \tanh(c_t)
\end{align*}
\]

A “candidate” hidden state that is computed based on the current input and the previous hidden state.
- It is exactly the same equation we had in our vanilla RNN!
- However, instead of taking the new hidden state as we did in the RNN, we will use the input gate from above to pick some of it.
Long-short-term-memories (LSTMs)

The internal memory of the unit combination of how we want to combine previous memory and the new input.

We could choose to ignore the old memory completely (forget gate all 0’s) or ignore the newly computed state completely (input gate all 0’s), but most likely we want something in between these two extremes.

\[
i_t = \sigma \left( W^{(i)} x_t + U^{(i)} h_{t-1} \right) \\
f_t = \sigma \left( W^{(f)} x_t + U^{(f)} h_{t-1} \right) \\
o_t = \sigma \left( W^{(o)} x_t + U^{(o)} h_{t-1} \right) \\
\hat{c}_t = \tanh \left( W^{(c)} x_t + U^{(c)} h_{t-1} \right) \\
c_t = f_t \circ c_{t-1} + i_t \circ \hat{c}_t \\
h_t = o_t \circ \tanh(c_t)
\]
Long-short-term-memories (LSTMs)

\[
i_t = \sigma \left( W^{(i)} x_t + U^{(i)} h_{t-1} \right) \\
f_t = \sigma \left( W^{(f)} x_t + U^{(f)} h_{t-1} \right) \\
o_t = \sigma \left( W^{(o)} x_t + U^{(o)} h_{t-1} \right) \\
\tilde{c}_t = \tanh \left( W^{(c)} x_t + U^{(c)} h_{t-1} \right) \\
c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t = o_t \odot \tanh(c_t)
\]

Given the memory \(c_t\), we finally compute the output hidden state \(h_t\) by multiplying the memory with the output gate.

Not all of the internal memory may be relevant to the hidden state used by other units in the network.
Illustrations a bit overwhelming ;)

Intuition: memory cells can keep information intact, unless inputs makes them forget it or overwrite it with new input. Cell can decide to output this information or just store it.
LSTMs are currently very hip!

- En vogue default model for most sequence labeling tasks
- Very powerful, especially when stacked and made even deeper (each hidden layer is already computed by a deep internal network)
- Most useful if you have lots and lots of data
Deep LSTMs don’t outperform traditional MT yet

<table>
<thead>
<tr>
<th>Method</th>
<th>test BLEU score (ntst14)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bahdanau et al. [2]</td>
<td>28.45</td>
</tr>
<tr>
<td>Baseline System [29]</td>
<td>33.30</td>
</tr>
<tr>
<td>Single forward LSTM, beam size 12</td>
<td>26.17</td>
</tr>
<tr>
<td>Single reversed LSTM, beam size 12</td>
<td>30.59</td>
</tr>
<tr>
<td>Ensemble of 5 reversed LSTMs, beam size 1</td>
<td>33.00</td>
</tr>
<tr>
<td>Ensemble of 2 reversed LSTMs, beam size 12</td>
<td>33.27</td>
</tr>
<tr>
<td>Ensemble of 5 reversed LSTMs, beam size 2</td>
<td>34.50</td>
</tr>
<tr>
<td>Ensemble of 5 reversed LSTMs, beam size 12</td>
<td>34.81</td>
</tr>
</tbody>
</table>

Table 1: The performance of the LSTM on WMT’14 English to French test set (ntst14). Note that an ensemble of 5 LSTMs with a beam of size 2 is cheaper than of a single LSTM with a beam of size 12.

<table>
<thead>
<tr>
<th>Method</th>
<th>test BLEU score (ntst14)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline System [29]</td>
<td>33.30</td>
</tr>
<tr>
<td>Cho et al. [5]</td>
<td>34.54</td>
</tr>
<tr>
<td>Best WMT’14 result [9]</td>
<td>37.0</td>
</tr>
<tr>
<td>Rescoring the baseline 1000-best with a single forward LSTM</td>
<td>35.61</td>
</tr>
<tr>
<td>Rescoring the baseline 1000-best with a single reversed LSTM</td>
<td>35.85</td>
</tr>
<tr>
<td>Rescoring the baseline 1000-best with an ensemble of 5 reversed LSTMs</td>
<td>36.5</td>
</tr>
<tr>
<td>Oracle Rescoring of the Baseline 1000-best lists</td>
<td>37.45</td>
</tr>
</tbody>
</table>
Deep LSTM for Machine Translation

PCA of vectors from last time step hidden layer

Sequence to Sequence Learning by Sutskever et al. 2014
Further Improvements: More Gates!

Gated Feedback Recurrent Neural Networks, Chung et al. 2015

(a) Conventional stacked RNN

(b) Gated Feedback RNN
LSTMs/GRU were designed to combat vanishing gradients through a *gating* mechanism.

### LSTM (1997)

\[
\begin{align*}
i & = \sigma(x_tU^i + s_{t-1}W^i) \\
f & = \sigma(x_tU^f + s_{t-1}W^f) \\
o & = \sigma(x_tU^o + s_{t-1}W^o) \\
g & = \tanh(x_tU^g + s_{t-1}W^g) \\
c_t & = c_{t-1} \odot f + g \odot i \\
s_t & = \tanh(c_t) \odot o
\end{align*}
\]

### GRU (2014)

\[
\begin{align*}
z & = \sigma(x_tU^z + s_{t-1}W^z) \\
r & = \sigma(x_tU^r + s_{t-1}W^r) \\
h & = \tanh(x_tU^h + (s_{t-1} \odot r)W^h) \\
s_t & = (1 - z) \odot h + z \odot s_{t-1}
\end{align*}
\]

A LSTM/GRU layer is just another way to compute a hidden state that was previously

\[
s_t = \tanh(Ux_t + Ws_{t-1})
\]
Summary

- Recurrent Neural Networks are powerful
- A lot of ongoing work right now
- Gated Recurrent Units even better
- LSTMs maybe even better (jury still out)
- This was an advanced lecture → gain intuition, encourage exploration