Deep Learning for Natural Language Processing

Kemal Kurniawan

Kata.ai Research Team

kemal@kata.ai

December 17, 2018

Overview

Introduction (very brief)

What is deep learning?

What is deep learning?

- Machine learning that uses deep neural networks
- Deep neural networks $\approx$ multi-layer neural networks

What is deep learning?

- Machine learning that uses deep neural networks
- Deep neural networks ≈ multi-layer neural networks

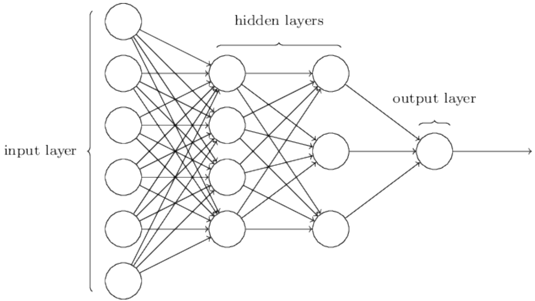


Figure: A feedforward neural network with 2 hidden layers.

(image from: `http://neuralnetworksanddeeplearning.com/`)

Neural net architectures for NLP

Neural net architectures for NLP

- Feedforward neural networks
  - Single layer or multiple layers
  - Fixed input and output size

Neural net architectures for NLP

- Feedforward neural networks
  - Single layer or multiple layers
  - Fixed input and output size
- Recurrent neural networks (RNN)
  - Characterized by a recurrent connection, i.e. previous output becomes (part of) current input
  - Parameter sharing over timesteps
  - Suitable for sequential inputs with varying length

Neural net architectures for NLP

- Feedforward neural networks
  - Single layer or multiple layers
  - Fixed input and output size
- Recurrent neural networks (RNN)
  - Characterized by a recurrent connection, i.e. previous output becomes (part of) current input
  - Parameter sharing over timesteps
  - Suitable for sequential inputs with varying length
- Convolutional neural networks (CNN)
  - Characterized by a convolution operator
  - Parameter sharing over image patches (CV) or n-grams (NLP)
  - Very successful in CV to incorporate *translation invariance*

Neural net architectures for NLP

- Feedforward neural networks
  - Single layer or multiple layers
  - Fixed input and output size
- Recurrent neural networks (RNN)
  - Characterized by a recurrent connection, i.e. previous output becomes (part of) current input
  - Parameter sharing over timesteps
  - Suitable for sequential inputs with varying length
- Convolutional neural networks (CNN)
  - Characterized by a convolution operator
  - Parameter sharing over image patches (CV) or n-grams (NLP)
  - Very successful in CV to incorporate *translation invariance*
- Recursive neural networks
  - Can operate on trees (e.g., syntax tree)
  - Parameter sharing over parent-children relations
  - Not so popular lately

Recurrent neural networks

Recurrent neural networks [Elman, 1990]

Recurrent neural networks [Elman, 1990]

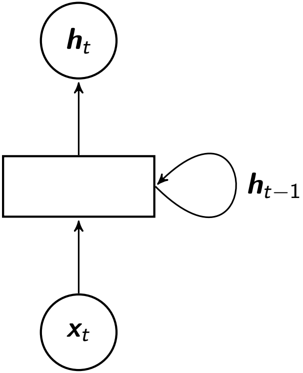


Figure: Architecture of an RNN. To compute the hidden state $\boldsymbol{h}_t$, the previous hidden state $\boldsymbol{h}_{t-1}$ is included as input along with $\boldsymbol{x}_t$.

[Kurniawan, 2017]

If we unfold an RNN, it looks similar to an MLP!

If we unfold an RNN, it looks similar to an MLP!

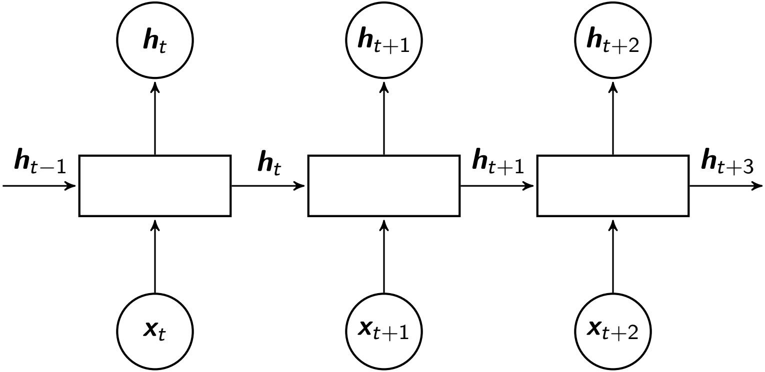


Figure: Architecture of an unfolded RNN for three time steps. This architecture is similar to that of a 3-layer feed-forward neural network.

[Kurniawan, 2017]

Mathematically, if $\boldsymbol{x}_t \in \mathbb{R}^d$ is an input vector at timestep t, then:

$$\boldsymbol{r}_t = \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{V}\boldsymbol{h}_{t-1} \quad (1)$$

$$\boldsymbol{h}_t = f\left(\boldsymbol{r}_t\right) \quad (2)$$

where $\boldsymbol{U} \in \mathbb{R}^{h \times d}$ and $\boldsymbol{V} \in \mathbb{R}^{h \times h}$ are parameters, and f is an activation function.

Mathematically, if $\boldsymbol{x}_t \in \mathbb{R}^d$ is an input vector at timestep t, then:

$$\boldsymbol{r}_t = \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{V}\boldsymbol{h}_{t-1} \tag{1}$$

$$\boldsymbol{h}_t = f\left(\boldsymbol{r}_t\right) \tag{2}$$

where $\boldsymbol{U} \in \mathbb{R}^{h \times d}$ and $\boldsymbol{V} \in \mathbb{R}^{h \times h}$ are parameters, and f is an activation function.

Output vector can be computed as:

$$\boldsymbol{s}_t = \boldsymbol{W}\boldsymbol{h}_t \tag{3}$$

$$\boldsymbol{y}_t = g\left(\boldsymbol{s}_t\right) \tag{4}$$

where $\boldsymbol{W} \in \mathbb{R}^{o \times d}$ is a parameter and g is a non-linear output function, usually depends on the task (e.g., softmax for classification).

Mathematically, if $\boldsymbol{x}_t \in \mathbb{R}^d$ is an input vector at timestep t, then:

$$\boldsymbol{r}_t = \boldsymbol{U}\boldsymbol{x}_t + \boldsymbol{V}\boldsymbol{h}_{t-1} \tag{1}$$
$$\boldsymbol{h}_t = f\left(\boldsymbol{r}_t\right) \tag{2}$$

where $\boldsymbol{U} \in \mathbb{R}^{h \times d}$ and $\boldsymbol{V} \in \mathbb{R}^{h \times h}$ are parameters, and f is an activation function.

Output vector can be computed as:

$$\boldsymbol{s}_t = \boldsymbol{W}\boldsymbol{h}_t \tag{3}$$
$$\boldsymbol{y}_t = g\left(\boldsymbol{s}_t\right) \tag{4}$$

where $\boldsymbol{W} \in \mathbb{R}^{o \times d}$ is a parameter and g is a non-linear output function, usually depends on the task (e.g., softmax for classification).

Note that $\boldsymbol{U}$, $\boldsymbol{V}$ and $\boldsymbol{W}$ are shared over timesteps!

Training an RNN

Training an RNN

- Initialize parameters $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{W}$ with small random numbers
- Start with an initial hidden state $\boldsymbol{h}_0$, which is usually set to a zero vector
- Perform forward computation to get output vectors $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots$ and compute the loss L

Training an RNN

- Initialize parameters $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{W}$ with small random numbers
- Start with an initial hidden state $\boldsymbol{h}_0$, which is usually set to a zero vector
- Perform forward computation to get output vectors $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots$ and compute the loss L
- Iteratively update RNN parameters $\boldsymbol{\theta}$ according to this rule

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial L}{\partial \boldsymbol{\theta}} \tag{5}$$

where η is a small positive number

Training an RNN

- Initialize parameters $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{W}$ with small random numbers
- Start with an initial hidden state $\boldsymbol{h}_0$, which is usually set to a zero vector
- Perform forward computation to get output vectors $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots$ and compute the loss L
- Iteratively update RNN parameters $\boldsymbol{\theta}$ according to this rule

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial L}{\partial \boldsymbol{\theta}} \tag{5}$$

  where η is a small positive number
- This is just *gradient descent*!
- A special backpropagation algorithm is required to compute gradients so RNN actually learns from early timesteps

Backpropagation through time (BPTT)

Backpropagation through time (BPTT)

At first, training an RNN with simple backpropagation seems straightforward. For example, if $L = \mathcal{L}(\boldsymbol{y}_t)$ denotes the loss of our RNN, the gradient on $\boldsymbol{W}$ seems to be:

$$\frac{\partial L}{\partial \boldsymbol{W}_{ij}} = \sum_k \frac{\partial L}{\partial \left(\boldsymbol{s}_t\right)_k} \frac{\partial \left(\boldsymbol{s}_t\right)_k}{\partial \boldsymbol{W}_{ij}} = \frac{\partial L}{\partial \left(\boldsymbol{s}_t\right)_i} \left(\boldsymbol{h}_t\right)_j \tag{6}$$

Backpropagation through time (BPTT)

At first, training an RNN with simple backpropagation seems straightforward. For example, if $L = \mathcal{L}(\boldsymbol{y}_t)$ denotes the loss of our RNN, the gradient on $\boldsymbol{W}$ seems to be:

$$\frac{\partial L}{\partial \boldsymbol{W}_{ij}} = \sum_k \frac{\partial L}{\partial \left(\boldsymbol{s}_t\right)_k} \frac{\partial \left(\boldsymbol{s}_t\right)_k}{\partial \boldsymbol{W}_{ij}} = \frac{\partial L}{\partial \left(\boldsymbol{s}_t\right)_i} \left(\boldsymbol{h}_t\right)_j \tag{6}$$

Or, in matrix notation:

$$\overline{\boldsymbol{W}} = \overline{\boldsymbol{s}}_t \boldsymbol{h}_t^\top \tag{7}$$

where $\overline{\boldsymbol{a}}$ denotes the gradient of L with respect to $\boldsymbol{a}$.

Backpropagation through time (BPTT)

At first, training an RNN with simple backpropagation seems straightforward. For example, if $L = \mathcal{L}(\boldsymbol{y}_t)$ denotes the loss of our RNN, the gradient on $\boldsymbol{W}$ seems to be:

$$\frac{\partial L}{\partial \boldsymbol{W}_{ij}} = \sum_k \frac{\partial L}{\partial \left(\boldsymbol{s}_t\right)_k} \frac{\partial \left(\boldsymbol{s}_t\right)_k}{\partial \boldsymbol{W}_{ij}} = \frac{\partial L}{\partial \left(\boldsymbol{s}_t\right)_i} \left(\boldsymbol{h}_t\right)_j \tag{6}$$

Or, in matrix notation:

$$\overline{\boldsymbol{W}} = \overline{\boldsymbol{s}}_t \boldsymbol{h}_t^\top \tag{7}$$

where $\overline{\boldsymbol{a}}$ denotes the gradient of L with respect to $\boldsymbol{a}$.

But this is **suboptimal**! Since $\boldsymbol{W}$ is shared across *all* timesteps, not just timestep t, it contributes to $\boldsymbol{s}_t$ for all timestep t. Thus, we need to sum the gradients from all timesteps, *if we want earlier inputs to have contribution to current outputs*.

If T denotes the maximum timestep, what we want is actually:

$$\overline{\boldsymbol{W}} = \sum_{t=1}^{T} \overline{\boldsymbol{s}}_t \boldsymbol{h}_t^\top \tag{8}$$

This is called *backpropagation through time* (BPTT).

If T denotes the maximum timestep, what we want is actually:

$$\overline{\boldsymbol{W}} = \sum_{t=1}^{T} \overline{\boldsymbol{s}}_t \boldsymbol{h}_t^\top \tag{8}$$

This is called *backpropagation through time* (BPTT).

$$\overline{\boldsymbol{U}} = \sum_{t=1}^{T} \overline{\boldsymbol{r}}_t \boldsymbol{x}_t^\top \tag{9}$$

$$\overline{\boldsymbol{V}} = \sum_{t=1}^{T} \overline{\boldsymbol{r}}_t \boldsymbol{h}_{t-1}^\top \tag{10}$$

$$\overline{\boldsymbol{r}}_t = \overline{\boldsymbol{h}}_t * f'(\boldsymbol{r}_t) \tag{11}$$

$$\overline{\boldsymbol{h}}_t = \boldsymbol{W}^\top \overline{\boldsymbol{s}}_t + \begin{cases} 0 & t = T \\ \boldsymbol{V}^\top \overline{\boldsymbol{r}}_{t+1} & \text{otherwise} \end{cases} \tag{12}$$

$$\overline{\boldsymbol{s}}_t = \overline{\boldsymbol{y}}_t * g'(\boldsymbol{s}_t) \tag{13}$$

A more detailed explanation on BPTT and how to implement it in code can be seen on `https://tinyurl.com/bptt-tutorial`.

[Goodfellow et al., 2016]

Vanishing/exploding gradient problem

Vanishing/exploding gradient problem

If we have no inputs $\boldsymbol{x}_t$ and a linear activation function f, Eq. 1 and 2 can be written as:

$$\boldsymbol{h}_t = \boldsymbol{V}\boldsymbol{h}_{t-1} \tag{14}$$

Vanishing/exploding gradient problem

If we have no inputs $\boldsymbol{x}_t$ and a linear activation function f, Eq. 1 and 2 can be written as:

$$\boldsymbol{h}_t = \boldsymbol{V}\boldsymbol{h}_{t-1} \tag{14}$$

If $\boldsymbol{V}$ is symmetric, it has an eigendecomposition $\boldsymbol{V} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$. Thus:

$$\boldsymbol{h}_t = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{h}_{t-1} \tag{15}$$

Vanishing/exploding gradient problem

If we have no inputs $\boldsymbol{x}_t$ and a linear activation function f, Eq. 1 and 2 can be written as:

$$\boldsymbol{h}_t = \boldsymbol{V}\boldsymbol{h}_{t-1} \tag{14}$$

If $\boldsymbol{V}$ is symmetric, it has an eigendecomposition $\boldsymbol{V} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$. Thus:

$$\boldsymbol{h}_t = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{h}_{t-1} \tag{15}$$

Applying this recursion repeatedly, we get:

$$\boldsymbol{h}_t = \boldsymbol{Q}\boldsymbol{\Lambda}^t\boldsymbol{Q}^\top \boldsymbol{h}_0 \tag{16}$$

Vanishing/exploding gradient problem

If we have no inputs $\boldsymbol{x}_t$ and a linear activation function f, Eq. 1 and 2 can be written as:

$$\boldsymbol{h}_t = \boldsymbol{V}\boldsymbol{h}_{t-1} \tag{14}$$

If $\boldsymbol{V}$ is symmetric, it has an eigendecomposition $\boldsymbol{V} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$. Thus:

$$\boldsymbol{h}_t = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top\boldsymbol{h}_{t-1} \tag{15}$$

Applying this recursion repeatedly, we get:

$$\boldsymbol{h}_t = \boldsymbol{Q}\boldsymbol{\Lambda}^t\boldsymbol{Q}^\top\boldsymbol{h}_0 \tag{16}$$

Eigenvalues with magnitude greater than one will explode; less than one will vanish. Elements of $\boldsymbol{h}_t$ not associated with the largest eigenvalue will be very small.

[Goodfellow et al., 2016]

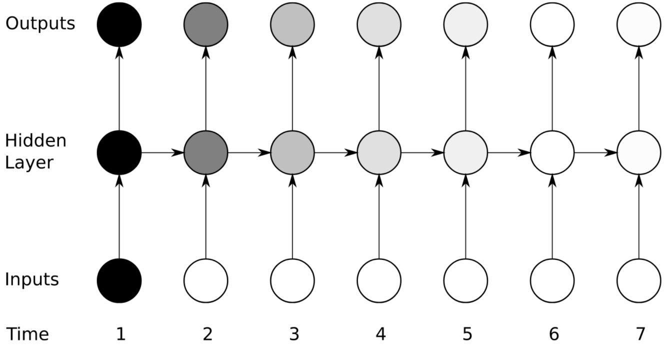


Figure: Vanishing gradient problem for RNN. The darker the shade, the greater the sensitivity of the nodes to the input at timestep one. [Graves, 2012]

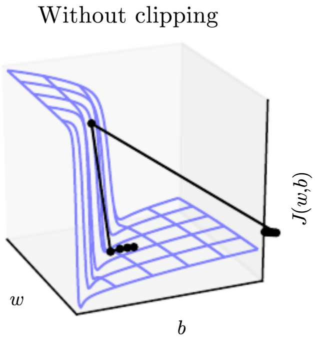


Figure: Illustration of the exploding gradient problem (left) and how gradient clipping might help (right). [Pascanu et al., 2013]

- In practice, gradients usually vanish rapidly for long sequences
- Thus, RNN cannot learn long-term dependency effectively
- To handle exploding gradients, typically gradient clipping is used: if $\|\boldsymbol{g}\| > v$ then

$$\boldsymbol{g} \leftarrow \frac{\boldsymbol{g}}{\|\boldsymbol{g}\|} v \tag{17}$$

- How to handle vanishing gradients?

Long short-term memory

Long short-term memory [Hochreiter and Schmidhuber, 1997]

Long short-term memory [Hochreiter and Schmidhuber, 1997]

- LSTM is a variant of RNN that is better equipped to handle the vanishing gradient problem
- Central to LSTM is the *cell state/memory cell*, which is a connection that runs through all timesteps
- LSTM can read from/write to the cell state freely, regulated by several *gates*
- We'll see more in the next few slides

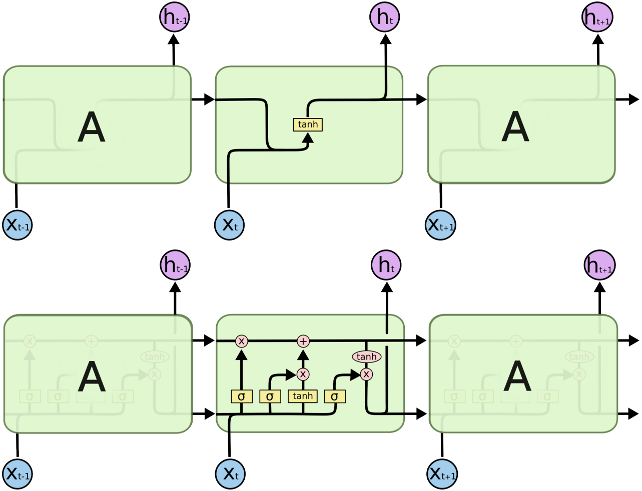
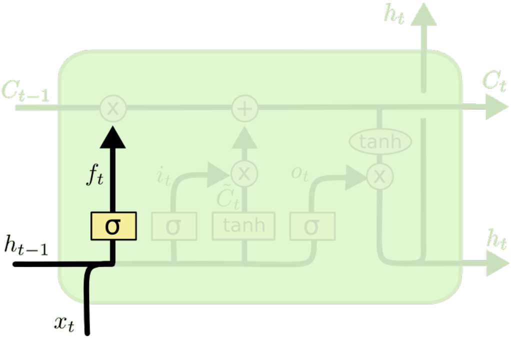


Figure: An unfolded RNN (top) and LSTM (bottom).

(image from: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`)

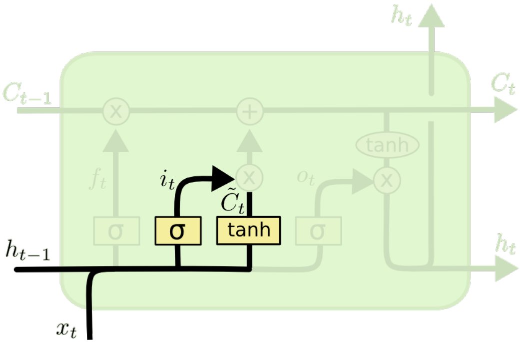
Forget gate



$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{U}_f \boldsymbol{x}_t + \boldsymbol{V}_f \boldsymbol{h}_{t-1}\right) \tag{18}$$

(image from: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`)

Input gate

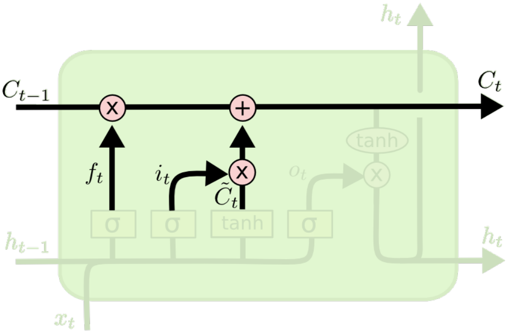


$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{U}_i \boldsymbol{x}_t + \boldsymbol{V}_i \boldsymbol{h}_{t-1}\right) \tag{19}$$

$$\tilde{\boldsymbol{C}}_t = \tanh\left(\boldsymbol{U}_C \boldsymbol{x}_t + \boldsymbol{V}_C \boldsymbol{h}_{t-1}\right) \tag{20}$$

(image from: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`)

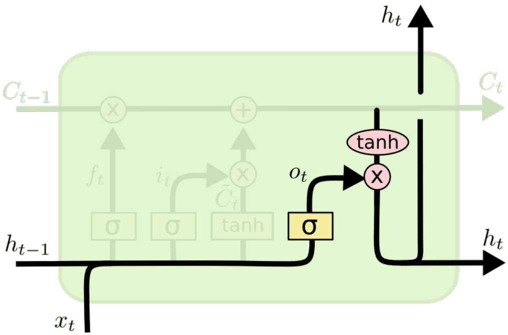
Cell state



$$\boldsymbol{C}_t = \boldsymbol{f}_t * \boldsymbol{C}_{t-1} + \boldsymbol{i}_t * \tilde{\boldsymbol{C}}_t \tag{21}$$

(image from: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`)

Output gate



$$o_t = \sigma\left(U_o x_t + V_o h_{t-1}\right) \tag{22}$$

$$h_t = o_t * \tanh(C_t) \tag{23}$$

(image from: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`)

Why LSTM?

Why LSTM?

- Address the vanishing gradient problem in RNN
  - The cell state acts as an "expressway", allowing gradients to flow easily
  - Reading/writing information to the cell state is regulated by the gates

Why LSTM?

- Address the vanishing gradient problem in RNN
  - The cell state acts as an "expressway", allowing gradients to flow easily
  - Reading/writing information to the cell state is regulated by the gates
- Very widely used and successful for many NLP tasks
  - POS tagging, NER, parsing, language modeling, machine translation, etc.
  - Vanilla RNN are almost never used in practice nowadays

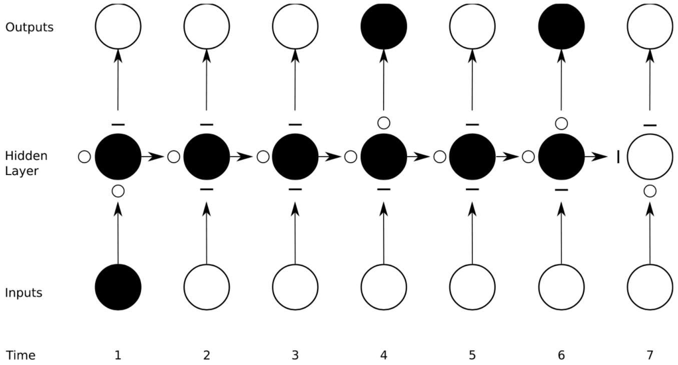


Figure: Preservation of gradient information by LSTM. For simplicity, gates are either open ('O') or closed ('–'). [Graves, 2012]

Example cases

Case 1: Character-level language modeling

Case 1: Character-level language modeling

- Task: given a sequence of words as history, predict the next word

Case 1: Character-level language modeling

- Task: given a sequence of words as history, predict the next word
- Formally, given word history $w_1, \ldots, w_t$, produce a probability distribution over next words:

$$\Pr\left(w_{t+1} \middle| w_t, \ldots, w_1\right) \tag{24}$$

Case 1: Character-level language modeling

- Task: given a sequence of words as history, predict the next word
- Formally, given word history $w_1, \ldots, w_t$, produce a probability distribution over next words:

$$\Pr\left(w_{t+1} | w_t, \ldots, w_1\right) \tag{24}$$

- N-gram language models approximate this probability by considering only the previous $n-1$ word as history, and represent the probability as multinomial distribution
- For example, with $n = 2$:

$$\Pr\left(w_{t+1} | w_t\right) = \frac{\text{count}\left(w_t, w_{t+1}\right) + \alpha}{\text{count}\left(w_t\right) + \alpha V} \tag{25}$$

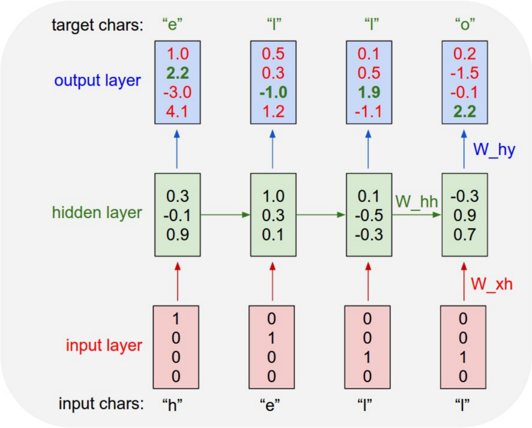
where α is a smoothing parameter and V is the vocabulary size

- Languages have long-term dependency; *the girl wearing a black dress and red shoes is pretty* $\rightarrow$ *is* depends on *girl*
- N-gram language models need larger value of n to capture long dependencies

- Languages have long-term dependency; *the girl wearing a black dress and red shoes is pretty* $\rightarrow$ *is* depends on *girl*
- N-gram language models need larger value of n to capture long dependencies
- **Issue 1**: the larger n is, the sparser our data becomes, i.e. the number of parameters to estimate grows exponentially and there is not enough data

- Languages have long-term dependency; *the girl wearing a black dress and red shoes is pretty* $\rightarrow$ *is* depends on *girl*
- N-gram language models need larger value of n to capture long dependencies
- **Issue 1**: the larger n is, the sparser our data becomes, i.e. the number of parameters to estimate grows exponentially and there is not enough data
- **Issue 2**: Smoothing parameter can prevent zero probabilities, but then all of them will have equal probability

- Languages have long-term dependency; *the girl wearing a black dress and red shoes is pretty* $\rightarrow$ *is* depends on *girl*
- N-gram language models need larger value of n to capture long dependencies
- **Issue 1**: the larger n is, the sparser our data becomes, i.e. the number of parameters to estimate grows exponentially and there is not enough data
- **Issue 2**: Smoothing parameter can prevent zero probabilities, but then all of them will have equal probability
- RNN-based language models can mitigate these issues!
- We'll see specifically character-level RNN-based language models, which operate on characters instead of words



An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters "hell" as input. The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o"); We want the green numbers to be high and red numbers to be low.

(image from: `https://karpathy.github.io/2015/05/21/rnn-effectiveness/`)

Generating text

Generating text

- Training: the correct next character is given as target output, and the model is trained to minimize negative log likelihood loss
- Start token `<INIT>` and end token `<EOS>` are added to training sentences

Generating text

- Training: the correct next character is given as target output, and the model is trained to minimize negative log likelihood loss
- Start token `<INIT>` and end token `<EOS>` are added to training sentences
- Generating: Feed `<INIT>` as input for the first timestep, select the most probable output as the next character, and feed that as the next input
- Generation stops when the model generates `<EOS>`
- We'll see the generation result of a model trained on Linux source code

Looks like a valid C code:

```c
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECONDS << 12;
  return segtable;
}
```

(image from: `https://karpathy.github.io/2015/05/21/rnn-effectiveness/`)

Copyright notice and include headers:

```
/*
 *  Copyright (c) 2006-2010, Intel Mobile Communications.  All rights reserved.
 *
 *   This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 as published by
 * the Free Software Foundation.
 *
 *        This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *
 *  GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *    along with this program; if not, write to the Free Software Foundation,
 *  Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/kexec.h>
#include <linux/errno.h>
#include <linux/io.h>
#include <linux/platform_device.h>
#include <linux/multi.h>
#include <linux/ckevent.h>

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/setew.h>
#include <asm/pgproto.h>
```

(image from: `https://karpathy.github.io/2015/05/21/rnn-effectiveness/`)

- The model can produce output that looks like a valid C code
- The output has nice indentations, matching brackets, comments, and even copyright notice and include headers
- But the model does not seem to “understand” the code: unused variables, undeclared variables, etc.

Case 2: Named-entity recognition

Case 2: Named-entity recognition

- Task: given a sentence, detect all spans of words corresponding to some entities (e.g., LOCATION, PERSON)

Case 2: Named-entity recognition

- Task: given a sentence, detect all spans of words corresponding to some entities (e.g., LOCATION, PERSON)
- Usually formulated as a sequence labeling problem, i.e. each word will be labeled as either (B)eginning, (I)nside, or (O)utside
- For example:

```
B-PER   O       O     B-LOC  I-LOC
Mark    lives   in    New    York
```

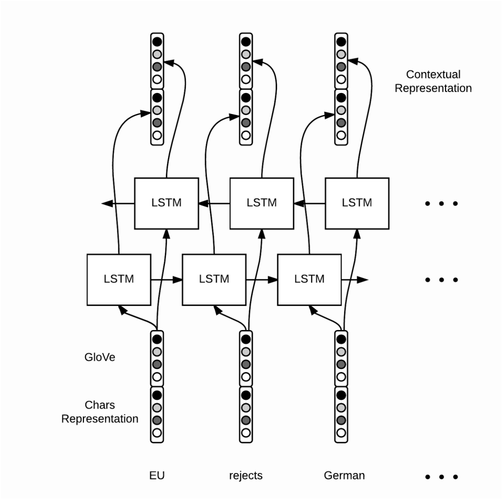
Case 2: Named-entity recognition

- Task: given a sentence, detect all spans of words corresponding to some entities (e.g., LOCATION, PERSON)
- Usually formulated as a sequence labeling problem, i.e. each word will be labeled as either (B)eginning, (I)nside, or (O)utside
- For example:

`B-PER`	`O`	`O`	`B-LOC`	`I-LOC`
`Mark`	`lives`	`in`	`New`	`York`

- A very common neural architecture for this task is based on [Lample et al., 2016]

Architecture

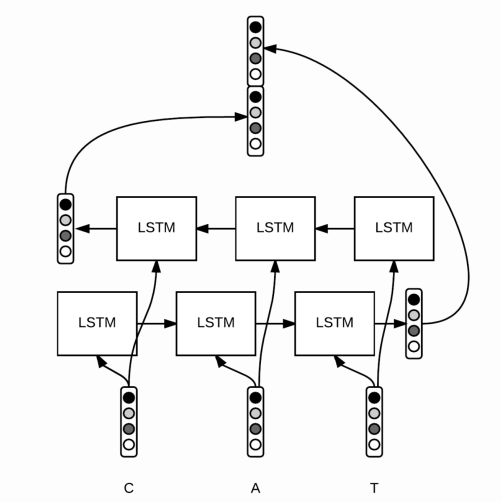


The output layer is a feedforward layer, followed by either a softmax or CRF layer.

(image from: `https://guillaumegenthial.github.io/sequence-tagging-with-tensorflow.html`)

Word representation from characters

Word representation from characters



(image from: `https://guillaumegenthial.github.io/sequence-tagging-with-tensorflow.html`)

Why use characters?

Why use characters?

- Character features are useful for NER (e.g., capitalization)

Why use characters?

- Character features are useful for NER (e.g., capitalization)
- Handles out-of-vocabulary (OOV) problem

Why use characters?

- Character features are useful for NER (e.g., capitalization)
- Handles out-of-vocabulary (OOV) problem
- Captures morphology to some extent

Results

Results

Model	$\mathbf{F_1}$
Collobert et al. (2011)*	89.59
Lin and Wu (2009)	83.78
Lin and Wu (2009)*	90.90
Huang et al. (2015)*	90.10
Passos et al. (2014)	90.05
Passos et al. (2014)*	90.90
Luo et al. (2015)* + gaz	89.9
Luo et al. (2015)* + gaz + linking	**91.2**
Chiu and Nichols (2015)	90.69
Chiu and Nichols (2015)*	90.77
LSTM-CRF (no char)	90.20
LSTM-CRF	**90.94**
S-LSTM (no char)	87.96
S-LSTM	90.33

Table 1: English NER results (CoNLL-2003 test set). * indicates models trained with the use of external labeled data

[Lample et al., 2016]

Results

Model	$\mathbf{F_1}$
Florian et al. (2003)*	72.41
Ando and Zhang (2005a)	75.27
Qi et al. (2009)	75.72
Gillick et al. (2015)	72.08
Gillick et al. (2015)*	76.22
LSTM-CRF – no char	75.06
LSTM-CRF	**78.76**
S-LSTM – no char	65.87
S-LSTM	75.66

Table 2: German NER results (CoNLL-2003 test set). * indicates models trained with the use of external labeled data

[Lample et al., 2016]

Using characters for OOV

Using characters for OOV

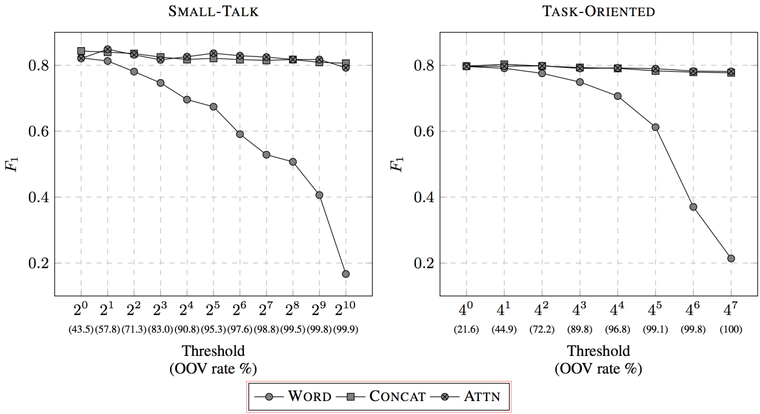


Figure 2: F_1 scores on the test set of each dataset with varying threshold. Words occurring fewer than this threshold in the training set are converted into the special token for OOV words. OOV rate increases as threshold does (from left to right). WORD, CONCAT, and ATTN refers to the word embedding-only, concatenation, and attention model respectively.

[Kurniawan and Louvan, 2018]

Case 3: Machine translation (and more)

Case 3: Machine translation (and more)

- Task: given a sentence $x_1, \ldots, x_N$ in the source language, find the translation $y_1, \ldots, y_M$ in the target language

Case 3: Machine translation (and more)

- Task: given a sentence $x_1, \ldots, x_N$ in the source language, find the translation $y_1, \ldots, y_M$ in the target language
- An RNN is trained to model

$$p\left(y_1, \ldots, y_M | x_1, \ldots, x_N\right) = \prod_{t=1}^{M} p_{\text{dec}}\left(y_t | y_1, \ldots, y_{t-1}, c\right) \tag{26}$$

  where $c = \text{RNN}_{\text{enc}}(\{x_1, \ldots, x_N\})$ and N may not equal M
- This model is usually called **sequence-to-sequence** or **encoder-decoder**
- Output layer is a softmax layer

Case 3: Machine translation (and more)

- Task: given a sentence $x_1, \ldots, x_N$ in the source language, find the translation $y_1, \ldots, y_M$ in the target language
- An RNN is trained to model

$$p\left(y_1, \ldots, y_M | x_1, \ldots, x_N\right) = \prod_{t=1}^{M} p_{\text{dec}}\left(y_t | y_1, \ldots, y_{t-1}, c\right) \tag{26}$$

  where $c = \text{RNN}_{\text{enc}}(\{x_1, \ldots, x_N\})$ and N may not equal M
- This model is usually called **sequence-to-sequence** or **encoder-decoder**
- Output layer is a softmax layer
- Training: the correct sentence pairs are given, and the model is trained to minimize negative log likelihood loss

Case 3: Machine translation (and more)

- Task: given a sentence $x_1, \ldots, x_N$ in the source language, find the translation $y_1, \ldots, y_M$ in the target language
- An RNN is trained to model

$$p\left(y_1, \ldots, y_M | x_1, \ldots, x_N\right) = \prod_{t=1}^{M} p_{\text{dec}}\left(y_t | y_1, \ldots, y_{t-1}, c\right) \tag{26}$$

  where $c = \text{RNN}_{\text{enc}}(\{x_1, \ldots, x_N\})$ and N may not equal M
- This model is usually called **sequence-to-sequence** or **encoder-decoder**
- Output layer is a softmax layer
- Training: the correct sentence pairs are given, and the model is trained to minimize negative log likelihood loss
- Decoding: decoder selects the most likely word at each timestep and feeds it back as input for the next timestep

[Sutskever et al., 2014]

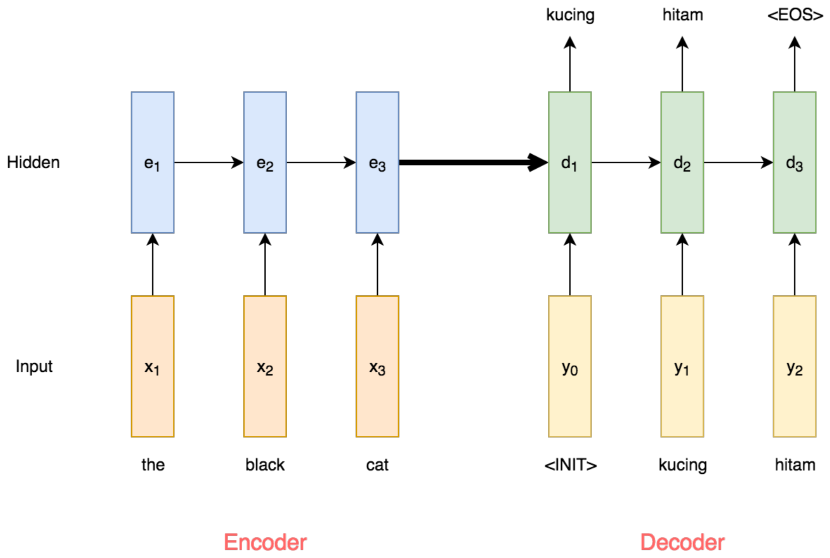
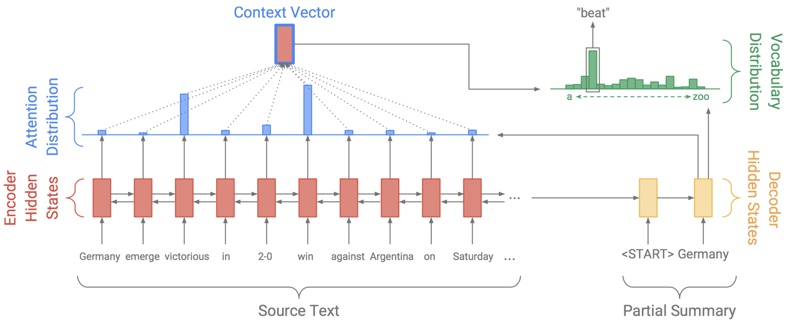


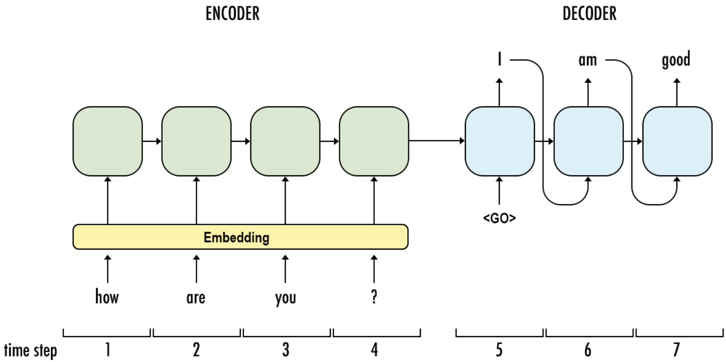
Figure: Illustration of an encoder-decoder network.

Encoder-decoder model can be used for many other tasks! For example, text summarization:



[See et al., 2017]

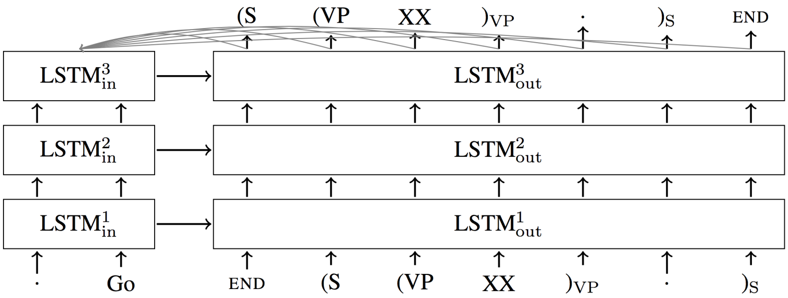
Dialog generation for open-domain dialog systems:



(image from:

https://towardsdatascience.com/sequence-to-sequence-model-introduction-and-concepts-44d9b41cd42d)

Even syntactic parsing:



[Vinyals et al., 2015]

Summary and Notes

Summary and Notes

- Deep learning is useful for NLP
- RNN is by far the most successful neural network architecture for NLP
- Watch out for vanishing/exploding gradient problem; use LSTM and gradient clipping!
- Don't worry too much about gradients; libraries like PyTorch, Keras, Tensorflow, etc. will compute them automatically
- Exciting stuff that aren't covered: attention, transfer learning, cross-lingual learning, etc.
- Start doing NLP **now**! Kata.ai is hiring!

Q & A

References I

Kemal Kurniawan (2017)
Exploring Recurrent Neural Network Grammars for Parsing Low-Resource Languages
MSc Thesis, University of Edinburgh.

Jeffrey L. Elman (1990)
Finding structure in time
Cognitive Science 14(2), 179 – 211.

Sepp Hochreiter and Jurgen Schmidhuber (1997)
Long short-term memory
Neural Computation 9(8), 1735 – 1780.

Alex Graves (2012)
Supervised Sequence Labeling with Recurrent Neural Networks
PhD Thesis, TU Munchen.

References II

Razvan Pascanu, Tomas Mikolov, Yoshua Bengio (2013)
On the Difficulty of Training Recurrent Neural Networks
Proceedings of ICML'13.

Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer (2016)
Neural Architectures for Named Entity Recognition
Proceedings of NAACL-HLT 2016, 260 – 270.

Kemal Kurniawan and Samuel Louvan (2018)
Empirical Evaluation of Character-Based Model on Neural Named-Entity Recognition in Indonesian Conversational Texts
Proceedings of the 2018 EMNLP Workshop W-NUT: The 4th Workshop on Noisy User-generated Text, 85 – 92.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville (2016)
Deep Learning
MIT Press

References III

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le (2014)
Sequence to Sequence Learning with Neural Networks
Advances in Neural Information Processing Systems

Abigail See, Peter J. Liu, Christopher D. Manning (2017)
Get to the point: Summarization with pointer-generator networks
Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)

Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton (2015)
Grammar as a Foreign Language
Advances in Neural Information Processing Systems