Sequencing Legal DNA

NLP for Law and Political Economy

8. Convolutions, Recurrence, and Attention

Reminder: Assignment 2 Due by April 15th

- Options:
  - Two reading response essays
  - one code replication exercise
  - problem set solution
  - one essay and solutions to 4 problems
  - COVID-19 task submission
- Upload to assignment dropbox (see syllabus).

Reminder: Assignment 2 Due by April 15th

- Options:
  - Two reading response essays
  - one code replication exercise
  - problem set solution
  - one essay and solutions to 4 problems
  - COVID-19 task submission
- Upload to assignment dropbox (see syllabus).
- Essays will be uploaded to the forum on April 16th/17th:
  - vote on some of your classsmates' essays, and comment on at least one, by Monday April 27th.

Sequence Data

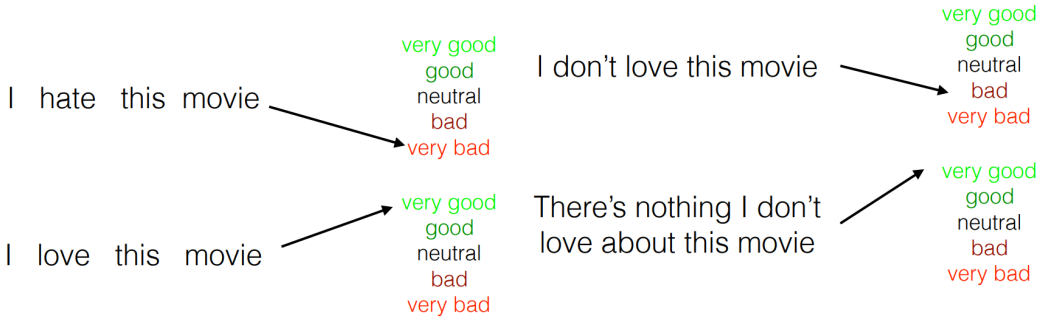
- This lecture focuses on deep learning approaches to NLP that treat language as a sequence of words.
- "Traditional" architectures:
  - Convolutional neural nets (CNNs)
  - Recurrent Neural Nets (RNNs)

Sequence Data

- This lecture focuses on deep learning approaches to NLP that treat language as a sequence of words.
- "Traditional" architectures:
  - Convolutional neural nets (CNNs)
  - Recurrent Neural Nets (RNNs)
- As of April 2020, CNNs and RNNs (as currently implemented) consistently get worse performance than attentional neural nets (transformers).

Outline

The Classic Sentence Classification Problem



Source: Graham Neubig slides.

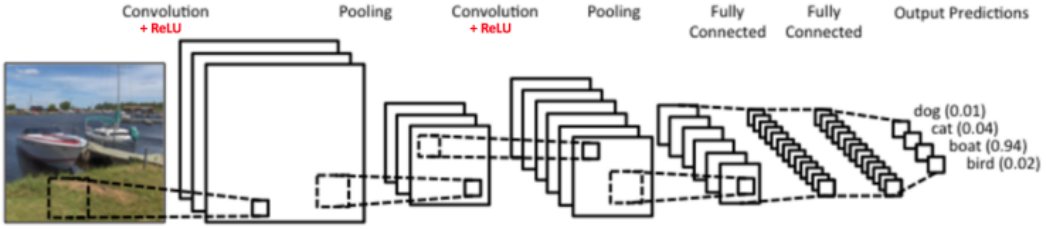
- (continuous) bag of words models (even with hidden layers) won't capture the importance of "don't love" or "nothing I don't love".

What we have done so far: N-Grams

What we have done so far: N-Grams

- Problems with n-grams models:
  - explosion in feature space
  - no sharing of information/weights across similar words/n-grams.

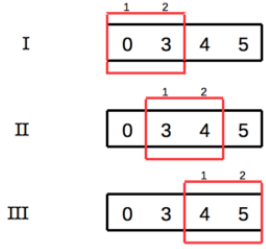
Convolutional neural nets



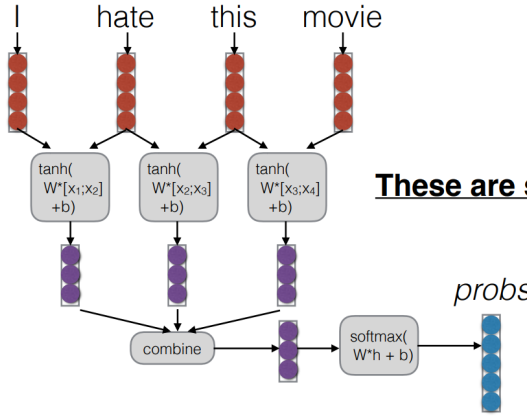
- A neural net architecture that constructs **filters** to extract **local structure** in data.
  - especially effective at image classification.

Sequence Convolution

- CNN's generate filters, such as the $\{\phi_1, \phi_2\} = \{1, 2\}$ here, and slides the filters across the input sequence.
- At each window, take the dot product:
  - `[0 3] · [1 2] = 6, [3 4] · [1 2] = 11, [4 5] · [1 2] = 14`
  - output $= \{6, 11, 14\}$



- CNN learns the weights for the filter $\{\phi_1, \phi_2\}$, to try to match the output
  - complicated CNNs have more filters, and different filter window sizes.



Graham Neubig slides.

Convolutional Neural Nets $\leftrightarrow$ N-gram Detectors

- More generally, can learn n_c-grams given a filter length n_c:

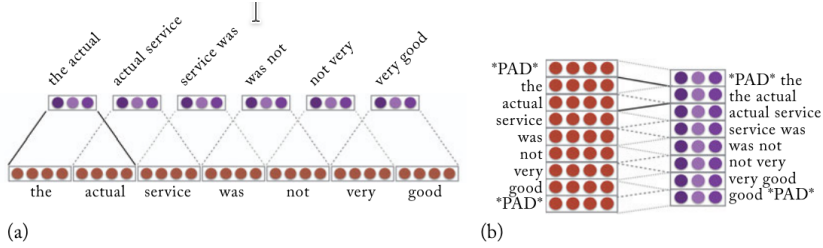


Figure 13.1: The inputs and outputs of a narrow and a wide convolution in the vector-concatenation and the vector-stacking notations. (a) A *narrow* convolution with a window of size $k = 2$ and 3-dimensional output ($\ell = 3$), in the vector-concatenation notation. (b) A *wide* convolution with a window of size $k = 2$, a 3-dimensional output ($\ell = 3$), in the vector-stacking notation.

- Let $w_{1:n_i}$ be a sequence of n_i words in document i, each with a n_E-dimensional embedding vector $\boldsymbol{w}_l$ contained in embedding matrix $\boldsymbol{E}$.
  - for CNN's, documents have to be the same length (can add padding tokens to short documents to achieve this.)

- Let $w_{1:n_i}$ be a sequence of n_i words in document i, each with a n_E-dimensional embedding vector $\boldsymbol{w}_l$ contained in embedding matrix $\boldsymbol{E}$.
  - for CNN's, documents have to be the same length (can add padding tokens to short documents to achieve this.)

- A 1D convolution of width n_c moves a **sliding window** of size n_c over the document:
  - Let $\boldsymbol{x}_l = [\boldsymbol{w}_l; \boldsymbol{w}_{l+1}; ...; \boldsymbol{w}_{l+n_c-1}]$ be the concatenated vectors for the n_c words in the window starting at index $l \in \{1, ..., n_i - n_c + 1\}$.

- Let $w_{1:n_i}$ be a sequence of n_i words in document i, each with a n_E-dimensional embedding vector $\boldsymbol{w}_l$ contained in embedding matrix $\boldsymbol{E}$.
  - for CNN's, documents have to be the same length (can add padding tokens to short documents to achieve this.)

- A 1D convolution of width n_c moves a **sliding window** of size n_c over the document:
  - Let $\boldsymbol{x}_l = [\boldsymbol{w}_l; \boldsymbol{w}_{l+1}; ...; \boldsymbol{w}_{l+n_c-1}]$ be the concatenated vectors for the n_c words in the window starting at index $l \in \{1, ..., n_i - n_c + 1\}$.

- The same **filter** is applied at each window:
  - a filter is a dot product with weight ϕ, usually followed by activation function $g(\cdot)$.
  - Formally:

$$h_l = g(\boldsymbol{x}_l \cdot \phi)$$

$$h_l \in \mathbb{R}, \boldsymbol{x}_l \in \mathbb{R}^{n_c n_E}, \phi \in \mathbb{R}^{n_c n_E}$$

- Let $w_{1:n_i}$ be a sequence of n_i words in document i, each with a n_E-dimensional embedding vector $\boldsymbol{w}_l$ contained in embedding matrix $\boldsymbol{E}$.
  - for CNN's, documents have to be the same length (can add padding tokens to short documents to achieve this.)

- A 1D convolution of width n_c moves a **sliding window** of size n_c over the document:
  - Let $\boldsymbol{x}_l = [\boldsymbol{w}_l; \boldsymbol{w}_{l+1}; ...; \boldsymbol{w}_{l+n_c-1}]$ be the concatenated vectors for the n_c words in the window starting at index $l \in \{1, ..., n_i - n_c + 1\}$.

- The same **filter** is applied at each window:
  - a filter is a dot product with weight ϕ, usually followed by activation function $g(\cdot)$.
  - Formally:

$$h_l = g(\boldsymbol{x}_l \cdot \phi)$$

$$h_l \in \mathbb{R}, \boldsymbol{x}_l \in \mathbb{R}^{n_c n_E}, \phi \in \mathbb{R}^{n_c n_E}$$

- Applied at each index l, generating a vector $\boldsymbol{h}$ with $n_i - n_c + 1$ values.
  - standard practice is to add special padding tokens at beginning and end of document, such that $\boldsymbol{h} \in \mathbb{R}^{n_i}$, the length of the document.

Convolutional layers have many filters

- Let n_ϕ be the number of filters, contained in a matrix ϕ.
- Then we have, for each token index l, a n_ϕ-vector

$$\boldsymbol{h}_l = \boldsymbol{g}(\boldsymbol{x}_l \cdot \boldsymbol{\phi})$$

$$\boldsymbol{h}_l \in \mathbb{R}^{n_\phi}, \boldsymbol{x}_l \in \mathbb{R}^{n_c n_E}, \phi \in \mathbb{R}^{n_c n_E \times n_\phi}$$

- Each dimension in ϕ (each filter) captures a different pattern – i.e., a predictive n-gram.

Convolutional layers have many filters

- Let n_ϕ be the number of filters, contained in a matrix ϕ.
- Then we have, for each token index l, a n_ϕ-vector

$$\boldsymbol{h}_l = \boldsymbol{g}(\boldsymbol{x}_l \cdot \boldsymbol{\phi})$$

$$\boldsymbol{h}_l \in \mathbb{R}^{n_\phi}, \boldsymbol{x}_l \in \mathbb{R}^{n_c n_E}, \phi \in \mathbb{R}^{n_c n_E \times n_\phi}$$

- Each dimension in ϕ (each filter) captures a different pattern – i.e., a predictive n-gram.
  - filters are initialized randomly; they shift toward different patterns (specialize) based on initial positions.

Convolutional layers have many filters

- Let n_ϕ be the number of filters, contained in a matrix ϕ.
- Then we have, for each token index l, a n_ϕ-vector

$$\boldsymbol{h}_l = \boldsymbol{g}(\boldsymbol{x}_l \cdot \boldsymbol{\phi})$$

$$\boldsymbol{h}_l \in \mathbb{R}^{n_\phi}, \boldsymbol{x}_l \in \mathbb{R}^{n_c n_E}, \phi \in \mathbb{R}^{n_c n_E \times n_\phi}$$

- Each dimension in ϕ (each filter) captures a different pattern – i.e., a predictive n-gram.
  - filters are initialized randomly; they shift toward different patterns (specialize) based on initial positions.
- Applied at each token index, the convolutional layer outputs a matrix $\boldsymbol{h} \in \mathbb{R}^{n_i \times n_\phi}$.

Pooling

- The convolutional layer outputs a matrix $\boldsymbol{h} \in \mathbb{R}^{n_i \times n_\phi}$.
- That matrix is then reduced to an n_ϕ-vector $\boldsymbol{s}_i$ for input to an MLP (e.g.) for regression or classification.
- This reduction is done with a **pooling layer**.

Pooling

- The convolutional layer outputs a matrix $\boldsymbol{h} \in \mathbb{R}^{n_i \times n_\phi}$.
- That matrix is then reduced to an n_ϕ-vector $\boldsymbol{s}_i$ for input to an MLP (e.g.) for regression or classification.
- This reduction is done with a **pooling layer**.
- The standard is "max pooling":
  - for each filter dimension $m \in \{1, ..., n_\phi\}$, look up the maximum value observed for that "feature" in the document:

$$\boldsymbol{s}_{[m]} = \max_{l \in \{1,...,n_i\}} \boldsymbol{h}_{[l,m]}, \forall m \in \{1, ..., n_\phi\}$$

    that is, across all token indexes l.

Pooling

- The convolutional layer outputs a matrix $\boldsymbol{h} \in \mathbb{R}^{n_i \times n_\phi}$.
- That matrix is then reduced to an n_ϕ-vector $\boldsymbol{s}_i$ for input to an MLP (e.g.) for regression or classification.
- This reduction is done with a **pooling layer**.
- The standard is "max pooling":
  - for each filter dimension $m \in \{1, ..., n_\phi\}$, look up the maximum value observed for that "feature" in the document:

$$\boldsymbol{s}_{[m]} = \max_{l \in \{1,...,n_i\}} \boldsymbol{h}_{[l,m]}, \forall m \in \{1, ..., n_\phi\}$$

    that is, across all token indexes l.

- can also take the average, but this usually gets worse performance.

Pooling

- The convolutional layer outputs a matrix $\boldsymbol{h} \in \mathbb{R}^{n_i \times n_\phi}$.
- That matrix is then reduced to an n_ϕ-vector $\boldsymbol{s}_i$ for input to an MLP (e.g.) for regression or classification.
- This reduction is done with a **pooling layer**.
- The standard is "max pooling":
  - for each filter dimension $m \in \{1, ..., n_\phi\}$, look up the maximum value observed for that "feature" in the document:

$$\boldsymbol{s}_{[m]} = \max_{l \in \{1,...,n_i\}} \boldsymbol{h}_{[l,m]}, \forall m \in \{1, ..., n_\phi\}$$

    that is, across all token indexes l.

- can also take the average, but this usually gets worse performance.
- Note that the vector $\boldsymbol{s}_i$ is a document embedding – documents with similar predictive information will have similar vectors $\boldsymbol{s}_i$.

Stride

- Can skip some token indexes, e.g., apply convolution to every other index.

Stride

- Can skip some token indexes, e.g., apply convolution to every other index.

Multiple Channels

- visual CNNs have multiple channels for different color axes: Red, Green, Blue.
  - can do the same thing in NLP: for example, one channel can have the words, a second could have the POS tags, and a third could have the dependency tags.

Stride

- Can skip some token indexes, e.g., apply convolution to every other index.

Multiple Channels

- visual CNNs have multiple channels for different color axes: Red, Green, Blue.
  - can do the same thing in NLP: for example, one channel can have the words, a second could have the POS tags, and a third could have the dependency tags.

Multiple Convolutions

- can have multiple parallel convolutional layers with different filter lengths n_c

Stride

- Can skip some token indexes, e.g., apply convolution to every other index.

Multiple Channels

- visual CNNs have multiple channels for different color axes: Red, Green, Blue.
  - can do the same thing in NLP: for example, one channel can have the words, a second could have the POS tags, and a third could have the dependency tags.

Multiple Convolutions

- can have multiple parallel convolutional layers with different filter lengths n_c

Dynamic Pooling

- Can do separate convolutions and/or pooling for different segments of the document.

Stride

- Can skip some token indexes, e.g., apply convolution to every other index.

Multiple Channels

- visual CNNs have multiple channels for different color axes: Red, Green, Blue.
  - can do the same thing in NLP: for example, one channel can have the words, a second could have the POS tags, and a third could have the dependency tags.

Multiple Convolutions

- can have multiple parallel convolutional layers with different filter lengths n_c

Dynamic Pooling

- Can do separate convolutions and/or pooling for different segments of the document.

Convolutions on the Parse Tree

- can follow parse tree branches, rather than use linear word order.

Instead of pooling and outputting to MLP, can output $\boldsymbol{h}_1$ to second convolutional layer.

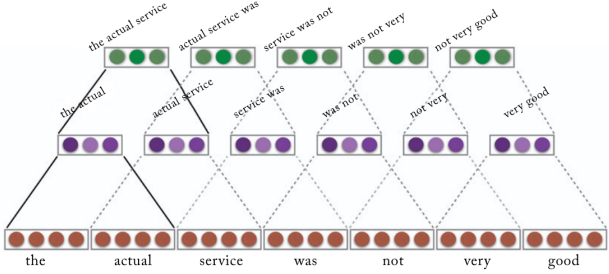


Figure 13.3: Two-layer hierarchical convolution with k=2.

Instead of pooling and outputting to MLP, can output $\boldsymbol{h}_1$ to second convolutional layer.

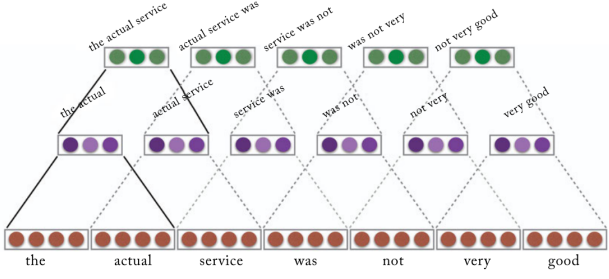


Figure 13.3: Two-layer hierarchical convolution with k=2.

- Let $\text{CONV}(x; c, m)$ be a 1D convolutional layer with m filters of window size c, applied to input (sequence of vectors) x.
- Can define arbitrarily deep stack of L convolutional layers:

$$\boldsymbol{h}_1 = \text{CONV}(\boldsymbol{w}_{1:n_i}; c_1, m_1)$$
$$\boldsymbol{h}_2 = \text{CONV}(\boldsymbol{h}_1; c_2, m_2)$$
$$\ldots$$
$$\boldsymbol{h}_L = \text{CONV}(\boldsymbol{h}_{L-1}; c_{L-1}, m_{L-1}$$
$$\boldsymbol{s} = \text{POOL}(\boldsymbol{h}_L; m_{L-1})$$

Johnson and Zhang 2015

- Use CNN for classifying documents by topic and sentiment.
- Interesting part is showing what CNN used for the prediction.

Johnson and Zhang 2015

- Use CNN for classifying documents by topic and sentiment.
- Interesting part is showing what CNN used for the prediction.
- Linear model (SVM) relied on single words and just a few n-grams:
  - poor, useless, returned, not worth, return, worse, disappointed, terrible, worst, horrible
  - great, excellent, perfect, love, easy, amazing, awesome, no problems, perfectly, beat

Johnson and Zhang 2015

- Use CNN for classifying documents by topic and sentiment.
- Interesting part is showing what CNN used for the prediction.

- Linear model (SVM) relied on single words and just a few n-grams:
  - poor, useless, returned, not worth, return, worse, disappointed, terrible, worst, horrible
  - great, excellent, perfect, love, easy, amazing, awesome, no problems, perfectly, beat
- CNN recovers longer, more interesting phrases:

N1	completely useless ., return policy .
N2	it won't even, but doesn't work
N3	product is defective, very disappointing !
N4	is totally unacceptable, is so bad
N5	was very poor, it has failed
P1	works perfectly !, love this product
P2	very pleased !, super easy to, i am pleased
P3	'm so happy, it works perfect, is awesome !
P4	highly recommend it, highly recommended !
P5	am extremely satisfied, is super fast

Table 5: Examples of predictive text regions in the training set.

were unacceptably bad, is abysmally bad, were universally poor, was hugely disappointed, was enormously disappointed, is monumentally frustrating, are endlessly frustrating
best concept ever, best ideas ever, best hub ever, am wholly satisfied, am entirely satisfied, am incredicbly satisfied, 'm overall impressed, am awfully pleased, am exceptionally pleased, 'm entirely happy, are acoustically good, is blindingly fast,

Table 6: Examples of text regions that contribute to prediction. They are from the *test set*, and they did *not* appear in the training set, either entirely or partially as bi-grams.

WaveNet

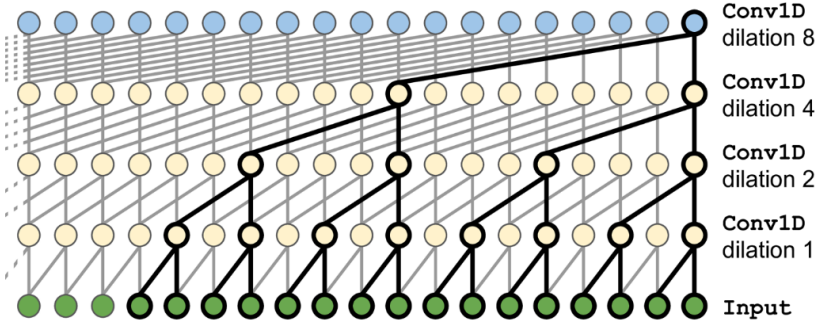


Figure 15-11. WaveNet architecture

- WaveNet is a 1-D CNN:
  - stacked convolution layers, doubling the dilation rate at every layer.

WaveNet

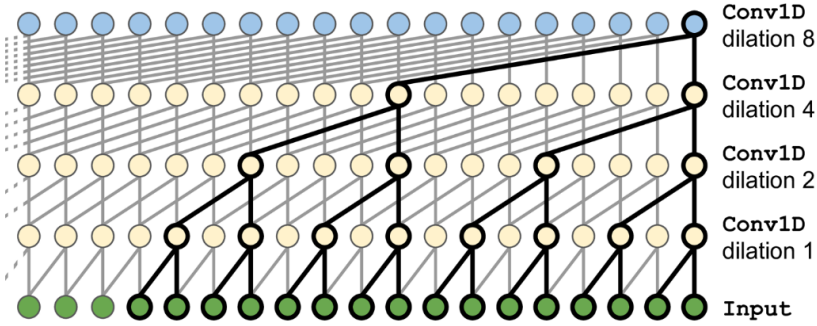
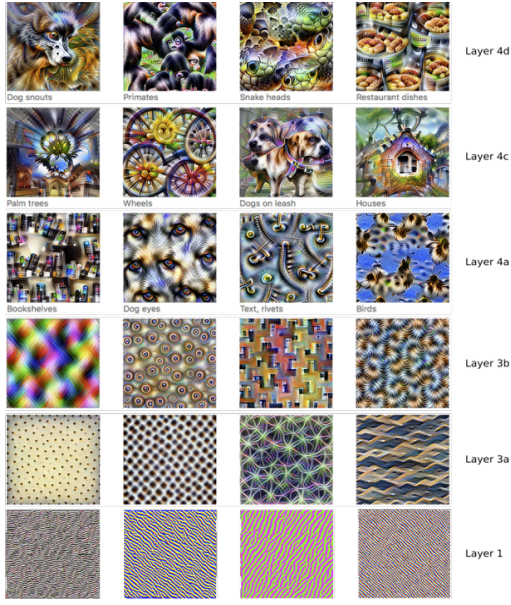


Figure 15-11. WaveNet architecture

- WaveNet is a 1-D CNN:
  - stacked convolution layers, doubling the dilation rate at every layer.
  - gets state-of-the-art performance on audio processing tasks, e.g. text to speech generation.
- For a similar NLP application, see Kalchenbrenner et al (2016).

How CNNs see the world



How CNNs see the world

Alternative: Embedded Hashed N-Grams

Alternative: Embedded Hashed N-Grams

Goldberg (2017) proposes the hashing vectorizer (**keras.preprocessing.text.hashing_trick**) as an efficient alternative to CNN's:

- Allocate $n_w \approx 1$ million rows to an embedding matrix $\boldsymbol{E}$
- Assign n-grams to embedding indexes with hashing function
- train MLP on top of embedding layer.

Alternative: Embedded Hashed N-Grams

Goldberg (2017) proposes the hashing vectorizer (**keras.preprocessing.text.hashing_trick**) as an efficient alternative to CNN's:

- Allocate $n_w \approx 1$ million rows to an embedding matrix $\boldsymbol{E}$
- Assign n-grams to embedding indexes with hashing function
- train MLP on top of embedding layer.
- Captures the local predictive power of n-grams without building vocabulary or costly training of CNN.

Outline

From vectors to sequences

- The models we have looked at so far took inputs of fixed dimensions across rows.

From vectors to sequences

- The models we have looked at so far took inputs of fixed dimensions across rows.
- RNNs work with sequences of arbitrary length:
  - can encode sequences into vectors.
  - can decode vectors in sequences.
    - therefore especially useful for language tasks such as translation.

From vectors to sequences

- The models we have looked at so far took inputs of fixed dimensions across rows.
- RNNs work with sequences of arbitrary length:
  - can encode sequences into vectors.
  - can decode vectors in sequences.
    - therefore especially useful for language tasks such as translation.
- state-of-the-art performance on some machine learning tasks, especially with short documents.

RNN

$$\boldsymbol{y}(\boldsymbol{x}_{\mathbf{1:}\boldsymbol{n}}) = \text{RNN}(\boldsymbol{x}_{\mathbf{1:}\boldsymbol{n}})$$

- $\boldsymbol{y}(\boldsymbol{x}_{\mathbf{1:}\boldsymbol{n}}) \in \mathbb{R}^y$, a label to predict, e.g. the topic of a news article.
- $\boldsymbol{x}_{\mathbf{1:}\boldsymbol{n}} = \boldsymbol{x}_1, \boldsymbol{x}_{2,}, ..., \boldsymbol{x}_n$ (embeddings of words $w_{1:n}$), $\boldsymbol{x}_l \in \mathbb{R}^{n_E}$
- a state vector $s_t \in \mathbb{R}^{n_s}$ at initial state $\boldsymbol{s}_0$ (usually zeros by convention)

RNN

$$\boldsymbol{y}(\boldsymbol{x_{1:n}}) = \text{RNN}(\boldsymbol{x_{1:n}})$$

- $\boldsymbol{y}(\boldsymbol{x_{1:n}}) \in \mathbb{R}^y$, a label to predict, e.g. the topic of a news article.
- $\boldsymbol{x_{1:n}} = \boldsymbol{x}_1, \boldsymbol{x}_{2,}, ..., \boldsymbol{x}_n$ (embeddings of words $w_{1:n}$), $\boldsymbol{x}_l \in \mathbb{R}^{n_E}$
- a state vector $s_t \in \mathbb{R}^{n_s}$ at initial state $\boldsymbol{s}_0$ (usually zeros by convention)
- At each step t:
  - a recursion function $R(s_{t-1}, x_t)$ computes the state vector s_t

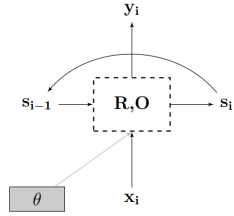
RNN

$$\boldsymbol{y}(\boldsymbol{x_{1:n}}) = \text{RNN}(\boldsymbol{x_{1:n}})$$

- $\boldsymbol{y}(\boldsymbol{x_{1:n}}) \in \mathbb{R}^y$, a label to predict, e.g. the topic of a news article.
- $\boldsymbol{x_{1:n}} = \boldsymbol{x}_1, \boldsymbol{x}_{2,}, ..., \boldsymbol{x}_n$ (embeddings of words $w_{1:n}$), $\boldsymbol{x}_l \in \mathbb{R}^{n_E}$
- a state vector $s_t \in \mathbb{R}^{n_s}$ at initial state $\boldsymbol{s}_0$ (usually zeros by convention)
- At each step t:
  - a recursion function $R(s_{t-1}, x_t)$ computes the state vector s_t
  - An output function $O(s_t)$ computes an output vector y_t (to be compared to the output data).

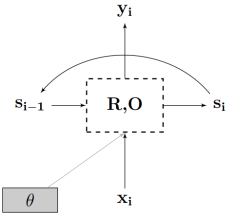
RNN Schema

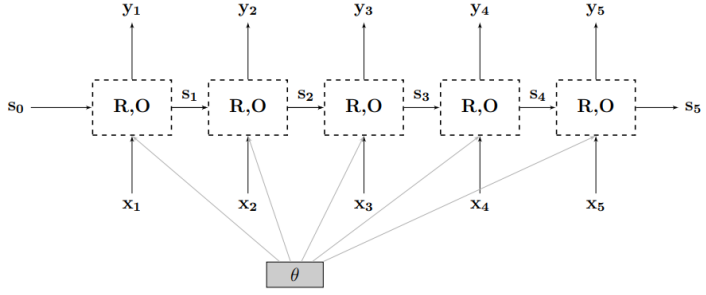
$$\hat{\boldsymbol{y}}_t = O(\boldsymbol{s}_t)$$
$$\boldsymbol{s}_t = R(\boldsymbol{s}_{t-1}, \boldsymbol{x}_t)$$



RNN Schema

$$\hat{\boldsymbol{y}}_t = O(\boldsymbol{s}_t)$$
$$\boldsymbol{s}_t = R(\boldsymbol{s}_{t-1}, \boldsymbol{x}_t)$$



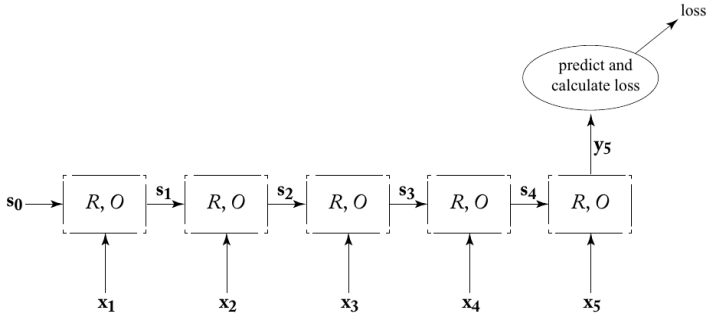


s_t and y_t encode the entire input sequence

$$
\begin{aligned}
\mathbf{s_4} &= R(\mathbf{s_3}, \mathbf{x_4}) \\
&= R(\overbrace{R(\mathbf{s_2}, \mathbf{x_3})}^{\mathbf{s_3}}, \mathbf{x_4}) \\
&= R(R(\overbrace{R(\mathbf{s_1}, \mathbf{x_2})}^{\mathbf{s_2}}, \mathbf{x_3}), \mathbf{x_4}) \\
&= R(R(R(\overbrace{R(\mathbf{s_0}, \mathbf{x_1})}^{\mathbf{s_1}}, \mathbf{x_2}), \mathbf{x_3}), \mathbf{x_4})
\end{aligned}
$$

RNN Usage

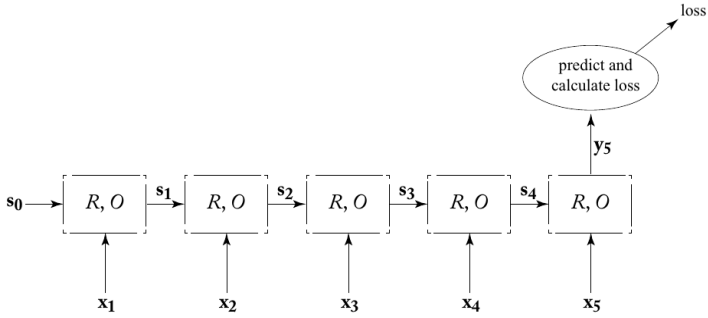
- **"Acceptor":** predict and calculate loss only for the last item in the sequence:



- **"Transducer":** compute the average loss for the output at every stage.

RNN Usage

- **"Acceptor":** predict and calculate loss only for the last item in the sequence:



- **"Transducer":** compute the average loss for the output at every stage.
- **"Encoder":** produces a vector for a sequence for input into a larger network/task.
  - e.g., translation into a sequence in another language.

Simple RNN

$$\hat{\boldsymbol{y}}_t = O(\boldsymbol{s}_t) = \boldsymbol{s}_t$$

$$\boldsymbol{s}_t = R(\boldsymbol{s}_{t-1}, \boldsymbol{x}_t) = \boldsymbol{g}(\boldsymbol{s}_{t-1} \cdot \omega_s + \boldsymbol{x}_t \cdot \omega_x)$$

$$\boldsymbol{s}_t, \boldsymbol{y}_t \in \mathbb{R}^{n_y}, x_t \in \mathbb{R}^{n_x}, \omega_s \in \mathbb{R}^{n_y \times n_y}, \omega_x \in \mathbb{R}^{n_x \times n_y}$$

- linear combination of state $\boldsymbol{s}_{t-1}$ and input $\boldsymbol{x}_t$ are passed through $g(\cdot)$ (typically ReLU or tanh) to get next state.
- output is just the state.

Simple RNN

$$\hat{\boldsymbol{y}}_t = O(\boldsymbol{s}_t) = \boldsymbol{s}_t$$

$$\boldsymbol{s}_t = R(\boldsymbol{s}_{t-1}, \boldsymbol{x}_t) = \boldsymbol{g}(\boldsymbol{s}_{t-1} \cdot \omega_s + \boldsymbol{x}_t \cdot \omega_x)$$

$$\boldsymbol{s}_t, \boldsymbol{y}_t \in \mathbb{R}^{n_y}, x_t \in \mathbb{R}^{n_x}, \omega_s \in \mathbb{R}^{n_y \times n_y}, \omega_x \in \mathbb{R}^{n_x \times n_y}$$

- linear combination of state $\boldsymbol{s}_{t-1}$ and input $\boldsymbol{x}_t$ are passed through $g(\cdot)$ (typically ReLU or tanh) to get next state.
- output is just the state.
- Simple RNN works well for simple NLP tasks
  - but anything that uses long-range dependencies will not work well because the state forgets information quickly.

Bidirectional RNNs

- train additional parallel RNN starting at end of document and moving backward.
- predicted output is function of both forward and backward states: $y_t = O(s_t^F, s_t^B)$

Bidirectional RNNs

- train additional parallel RNN starting at end of document and moving backward.
- predicted output is function of both forward and backward states: $y_t = O(s_t^F, s_t^B)$

Stacked RNNs

- RNN's can be stacked, with multiple layers at each period.

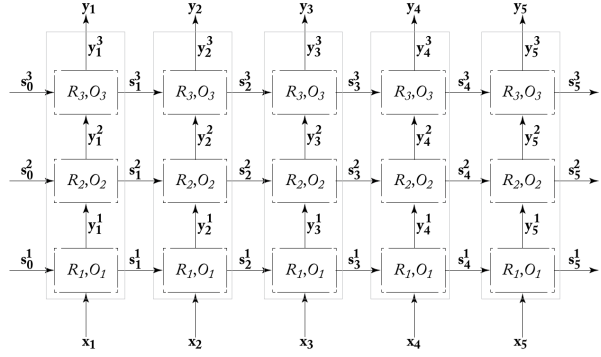


Figure 14.7: A three-layer ("deep") RNN architecture.

Gated Architectures – LSTM (Long Short-Term Memory)

- Inputs $\boldsymbol{x}_t \in \mathbb{R}^{n_x}$
- State vector has two memory components:
  - long term $\boldsymbol{s}_t^L \in \mathbb{R}^{n_y}$ and short term $\boldsymbol{s}_t^S \in \mathbb{R}^{n_y}$

Gated Architectures – LSTM (Long Short-Term Memory)

- Inputs $\boldsymbol{x}_t \in \mathbb{R}^{n_x}$
- State vector has two memory components:
  - long term $\boldsymbol{s}_t^L \in \mathbb{R}^{n_y}$ and short term $\boldsymbol{s}_t^S \in \mathbb{R}^{n_y}$
- Output:

$$\boldsymbol{y}_t = \boldsymbol{s}_t^S = g_O(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}^S) \otimes \tanh(\boldsymbol{s}_t^L)$$

  - output gate: $g_O = \text{sigmoid}(\boldsymbol{x}_t \cdot \omega_x^O + \boldsymbol{s}_{t-1}^S \cdot \omega_s^O)$
    - $\otimes$ = element-wise multiplication

Gated Architectures – LSTM (Long Short-Term Memory)

- Inputs $\boldsymbol{x}_t \in \mathbb{R}^{n_x}$
- State vector has two memory components:
  - long term $\boldsymbol{s}_t^L \in \mathbb{R}^{n_y}$ and short term $\boldsymbol{s}_t^S \in \mathbb{R}^{n_y}$
- Output:

$$\boldsymbol{y}_t = \boldsymbol{s}_t^S = g_O(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}^S) \otimes \tanh(\boldsymbol{s}_t^L)$$

  - output gate: $g_O = \text{sigmoid}(\boldsymbol{x}_t \cdot \omega_x^O + \boldsymbol{s}_{t-1}^S \cdot \omega_s^O)$
    - $\otimes$ = element-wise multiplication

- Long-Term Memory Update:

$$s_t^L = g_F(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}^S) \otimes \boldsymbol{s}_{t-1}^L + g_I(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}^S) \otimes g_U(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}^S)$$

  - forget gate: $g_F = \text{sigmoid}(\boldsymbol{x}_t \cdot \omega_x^F + \boldsymbol{s}_{t-1}^S \cdot \omega_s^F)$
  - input gate: $g_I = \text{sigmoid}(\boldsymbol{x}_t \cdot \omega_x^I + \boldsymbol{s}_{t-1}^S \cdot \omega_s^I)$
  - state updater: $g_U = \tanh(\boldsymbol{x}_t \cdot \omega_x^U + \boldsymbol{s}_{t-1}^S \cdot \omega_s^U)$

Gated Architectures – LSTM (Long Short-Term Memory)

- Inputs $\boldsymbol{x}_t \in \mathbb{R}^{n_x}$
- State vector has two memory components:
  - long term $\boldsymbol{s}_t^L \in \mathbb{R}^{n_y}$ and short term $\boldsymbol{s}_t^S \in \mathbb{R}^{n_y}$
- Output:

$$\boldsymbol{y}_t = \boldsymbol{s}_t^S = g_O(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}^S) \otimes \tanh(\boldsymbol{s}_t^L)$$

  - output gate: $g_O = \text{sigmoid}(\boldsymbol{x}_t \cdot \omega_x^O + \boldsymbol{s}_{t-1}^S \cdot \omega_s^O)$
    - $\otimes$ = element-wise multiplication

- Long-Term Memory Update:

$$s_t^L = g_F(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}^S) \otimes \boldsymbol{s}_{t-1}^L + g_I(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}^S) \otimes g_U(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}^S)$$

  - forget gate: $g_F = \text{sigmoid}(\boldsymbol{x}_t \cdot \omega_x^F + \boldsymbol{s}_{t-1}^S \cdot \omega_s^F)$
  - input gate: $g_I = \text{sigmoid}(\boldsymbol{x}_t \cdot \omega_x^I + \boldsymbol{s}_{t-1}^S \cdot \omega_s^I)$
  - state updater: $g_U = \tanh(\boldsymbol{x}_t \cdot \omega_x^U + \boldsymbol{s}_{t-1}^S \cdot \omega_s^U)$

- gating mechanisms prevent vanishing/exploding gradients.

GRU (Gated Recurrent Unit)

- GRU is a simplified LSTM with fewer parameters but similar performance (Cho et al 2014; Chung et al 2014).
- Does not separate long/short term memory, so just has a state vector $\boldsymbol{s}_t \in \mathbb{R}^{n_y}$
- Output:

$$\boldsymbol{y}_t = \boldsymbol{s}_t$$

GRU (Gated Recurrent Unit)

- GRU is a simplified LSTM with fewer parameters but similar performance (Cho et al 2014; Chung et al 2014).
- Does not separate long/short term memory, so just has a state vector $\boldsymbol{s}_t \in \mathbb{R}^{n_y}$
- Output:

$$\boldsymbol{y}_t = \boldsymbol{s}_t$$

- State Update:

$$\boldsymbol{s}_t = (1 - g_F(\boldsymbol{x}_t, \boldsymbol{s}_{t-1})) \otimes \boldsymbol{s}_{t-1} + g_F(\boldsymbol{x}_t, \boldsymbol{s}_{t-1}) \otimes g_U(\boldsymbol{x}_t, \boldsymbol{s}_{t-1})$$

  - forget gate: $g_F = \text{sigmoid}(\boldsymbol{x}_t \cdot \omega_x^F + \boldsymbol{s}_{t-1} \cdot \omega_s^F)$
  - update proposal: $g_U = \tanh(\boldsymbol{x}_t \cdot \omega_x^s + (\text{sigmoid}(\boldsymbol{x}_t \cdot \omega_x^U + \boldsymbol{s}_{t-1} \cdot \omega_s^U) \otimes \boldsymbol{s}_{t-1}) \cdot \omega_s^s)$

Dropout / Normalization in RNNs

- Gal (2015):
  - when applying dropout to RNNs, use same mask at all t.
  - droput mask sampled once per sequence, not once per time period.

Dropout / Normalization in RNNs

- Gal (2015):
  - when applying dropout to RNNs, use same mask at all t.
  - droput mask sampled once per sequence, not once per time period.
- Batch normalization doesn't work well for RNN's.
- Instead, can use **layer normalization**:
  - normalize each instace across feature dimensions
  - (rather than normalizing each feature across instances in the batch, as in BN)

Combining CNN's and RNN's

- Geron (Ch. 15, pp. 520-521) suggests the following hybrid network for sequence-to-vector tasks:
  1. four-word convolutional layer with twenty filters and a stride of two.
  2. GRU on top of the twenty filters
- Actually gets best performance on a sample task forecasting stock market prices.

RNNs: Practical Use for Sequence-to-Vector Task

For example, sentiment analysis:

- tokenize the documents into $\boldsymbol{w}_{1:n}$
- embedding $\boldsymbol{x}_t = \boldsymbol{w}_t \cdot \boldsymbol{E}$
  - embedding matrix $\boldsymbol{E}$ can be initialized with pre-trained GloVe embeddings.
- bidirectional RNN (e.g. LSTM) on $\boldsymbol{x_{1:n}}$ (and $\boldsymbol{x_{n:1}}$) to generate document vector $\boldsymbol{s}$
- MLP takes $\boldsymbol{s}$ to predict $\boldsymbol{y}$

RNNs: Practical Use for Sequence-to-Vector Task

For example, sentiment analysis:

- tokenize the documents into $\boldsymbol{w}_{1:n}$
- embedding $\boldsymbol{x}_t = \boldsymbol{w}_t \cdot \boldsymbol{E}$
  - embedding matrix $\boldsymbol{E}$ can be initialized with pre-trained GloVe embeddings.
- bidirectional RNN (e.g. LSTM) on $\boldsymbol{x}_{\boldsymbol{1:n}}$ (and $\boldsymbol{x}_{\boldsymbol{n:1}}$) to generate document vector $\boldsymbol{s}$
- MLP takes $\boldsymbol{s}$ to predict $\boldsymbol{y}$

keras functional API makes this straightforward.

RNNs: Practical tips for avoiding gradient problems

Richard Socher lecture notes

- Initialization:
  - recurrent matrices to be orthogonal
  - other matrices to have small scale
  - forget gate to 1 (default to remember)
- Learning rate:
  - use adaptive learning rate, e.g. Adam
- Gradient clipping: clip the norm, 1-5 should be reasonable threshold.

RNN's for predicting partisanship

Iyyer, Enns, Boyd-Graber, and Resnik (2014)

RNN's for predicting partisanship

Iyyer, Enns, Boyd-Graber, and Resnik (2014)

n	Most conservative n-grams	Most liberal n-grams
1	Salt, Mexico, housework, speculated, consensus, lawyer, pharmaceuticals, ruthless, deadly, Clinton, redistribution	rich, antipsychotic, malaria, biodiversity, richest, gene, pesticides, desertification, Net, wealthiest, labor, fertilizer, nuclear, HIV
3	prize individual liberty, original liberal idiots, stock market crash, God gives freedom, federal government interference, federal oppression nullification, respect individual liberty, Tea Party patriots, radical Sunni Islamists, Obama stimulus programs	rich and poor,"corporate greed", super rich pay, carrying the rich, corporate interest groups, young women workers, the very rich, for the rich, by the rich, soaking the rich, getting rich often, great and rich, the working poor, corporate income tax, the poor migrants
5	spending on popular government programs, bailouts and unfunded government promises, North America from external threats, government regulations place on businesses, strong Church of Christ convictions, radical Islamism and other threats	the rich are really rich, effective forms of worker participation, the pensions of the poor, tax cuts for the rich, the ecological services of biodiversity, poor children and pregnant women, vacation time for overtime pay
7	government intervention helped make the Depression Great, by God in His image and likeness, producing wealth instead of stunting capital creation, the traditional American values of limited government, trillions of dollars to overseas oil producers, its troubled assets to federal sugar daddies, Obama and his party as racialist fanatics	African Americans and other disproportionately poor groups; the growing gap between rich and poor; the Bush tax cuts for the rich; public outrage at corporate and societal greed; sexually transmitted diseases , most notably AIDS; organize unions or fight for better conditions, the biggest hope for health care reform

Table 2: Highest probability n-grams for conservative and liberal ideologies, as predicted by the **RNN2-(W2V)** model.

Predicting partisanship: Disagreement Analysis

Iyyer, Enns, Boyd-Graber, and Resnik (2014)

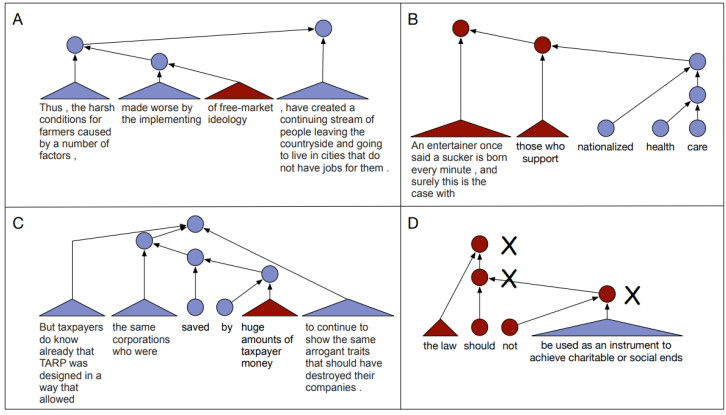


Figure 5: Predictions by **RNN2-(W2V)** on four sentences from the IBC. Node color is the true label (red for conservative, blue for liberal), and an "X" next to a node means the model's prediction was wrong. In *A* and *C*, the model accurately detects conservative-to-liberal polarity switches, while in *B* it correctly predicts the liberal-to-conservative switch. In *D*, negation confuses our model.

- A, B, and C were all mis-classified by standard bag-of-ngrams models. D tricks all the models.

Vectors and Sequences

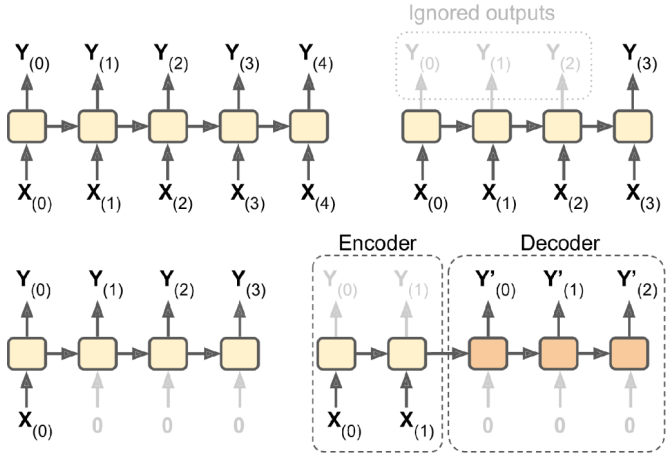


Figure 15-4. Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and Encoder–Decoder (bottom right) networks

- top left: sequence to sequence; top right: sequence to vector; bottom left: vector to sequence.
- bottom right: encoder-decoder.

Recurrent Autoencoders

To compress text (or other sequential) data, can use a recurrent autoencoder:

1. encode a sequence using a sequence-to-vector RNN, to compress the sequence to a single vector
2. decode the vector using a vector-to-sequence RNN to do the opposite.

Recurrent Autoencoders

To compress text (or other sequential) data, can use a recurrent autoencoder:

1. encode a sequence using a sequence-to-vector RNN, to compress the sequence to a single vector
2. decode the vector using a vector-to-sequence RNN to do the opposite.

- Reconstructions are usually quite bad.

Training an RNN Translator

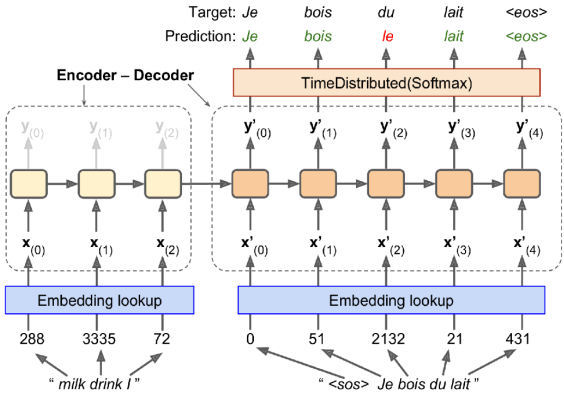


Figure 16-3. A simple machine translation model

1. English sentences are input to the encoder in reverse.
2. Decoder takes the encoded vector and the true French sentence, lagged by one word.
3. Output layer is softmax over French vocabulary.

Using the Translator

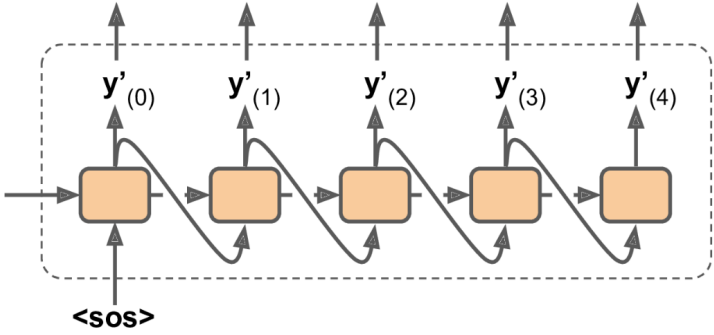


Figure 16-4. Feeding the previous output word as input at inference time

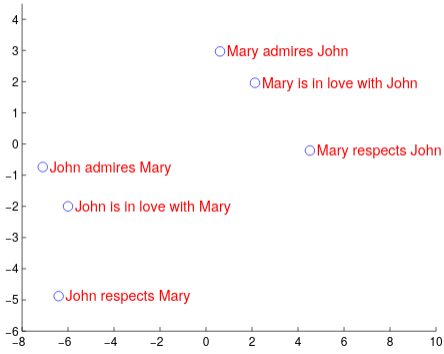
- When using the machine for translation, give it the start-of-sentence tag to get started, then feed in the generated words.

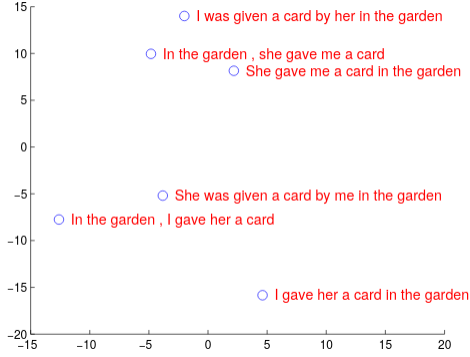
Geometry of Embedded Sequences

- Like CNN's, RNN's produce document vectors $\boldsymbol{s}$ with interpretable geometry.

Geometry of Embedded Sequences

- Like CNN's, RNN's produce document vectors $\boldsymbol{s}$ with interpretable geometry.





Sutskever, Vinyals, and Le, "Sequence to sequence learning with neural networks."

Outline

Outline

Rationalizing Neural Predictions

- Lei, Barzilay, and Jaakola (2016) provide a text-based prediction model which learns to extract snippets of text to serve as justifications – rationales.

Rationalizing Neural Predictions

- Lei, Barzilay, and Jaakola (2016) provide a text-based prediction model which learns to extract snippets of text to serve as justifications – rationales.

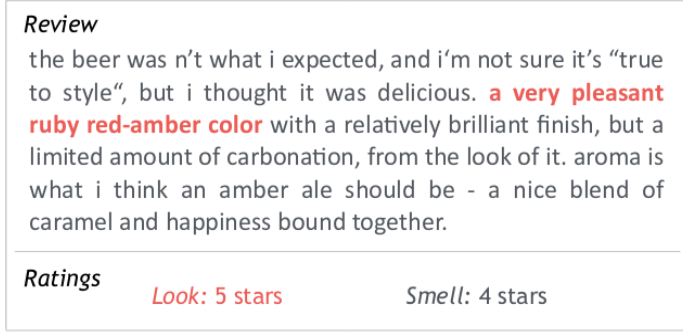


Figure 1: An example of a review with ranking in two categories. The rationale for Look prediction is shown in bold.

- Key idea: find minimal span(s) of text that can (by themselves) explain the prediction

Architecture

Lei, Barzilay, and Jaakola (2016)

- $\boldsymbol{x}$ = input text sequence, $\boldsymbol{y}$ = output class

Architecture

Lei, Barzilay, and Jaakola (2016)

- $\boldsymbol{x}$ = input text sequence, $\boldsymbol{y}$ = output class
- Encoder network:
  - takes the output of the generator, inputs to an RNN (e.g. bidirectional GRU) to predict the outcome from the text.

Architecture

Lei, Barzilay, and Jaakola (2016)

- $\boldsymbol{x}$ = input text sequence, $\boldsymbol{y}$ = output class
- Encoder network:
  - takes the output of the generator, inputs to an RNN (e.g. bidirectional GRU) to predict the outcome from the text.
- Generator network:
  - outputs a probability distribution for sub-sequences being rationales – snippets of text that are sent to the encoder network.

Architecture

Lei, Barzilay, and Jaakola (2016)

- $\boldsymbol{x}$ = input text sequence, $\boldsymbol{y}$ = output class
- Encoder network:
  - takes the output of the generator, inputs to an RNN (e.g. bidirectional GRU) to predict the outcome from the text.
- Generator network:
  - outputs a probability distribution for sub-sequences being rationales – snippets of text that are sent to the encoder network.
  - in conjunction with the encoder, works to extract a subset of text that is as predictive as the full text.

Architecture

Lei, Barzilay, and Jaakola (2016)

- $\boldsymbol{x}$ = input text sequence, $\boldsymbol{y}$ = output class
- Encoder network:
  - takes the output of the generator, inputs to an RNN (e.g. bidirectional GRU) to predict the outcome from the text.
- Generator network:
  - outputs a probability distribution for sub-sequences being rationales – snippets of text that are sent to the encoder network.
  - in conjunction with the encoder, works to extract a subset of text that is as predictive as the full text.
- Joint objective:
  - predicting $\boldsymbol{y}$ correctly.
  - regularizer that penalizes many/long rationales.

Sentiments about Beer

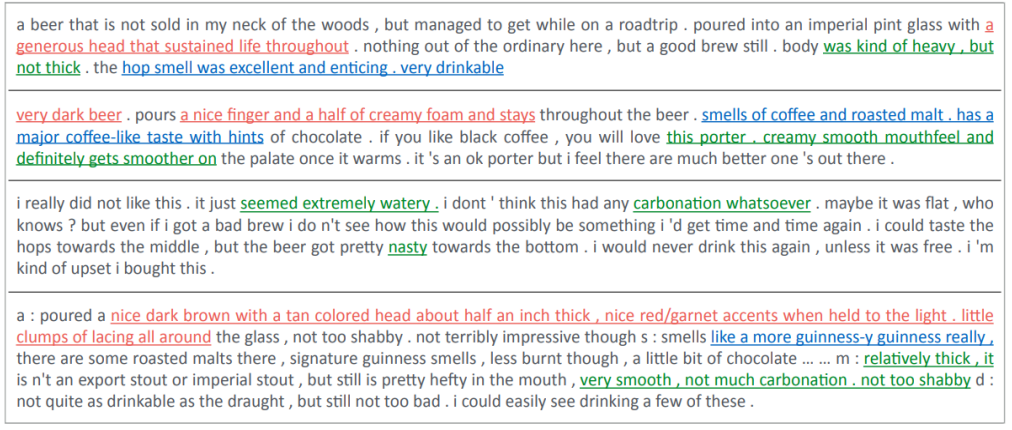


Figure 3: Examples of extracted rationales indicating the sentiments of various aspects. The extracted texts for appearance, smell and palate are shown in red, blue and green color respectively. The last example is shortened for space.

Relevant Features for Identifying Duplicate Questions

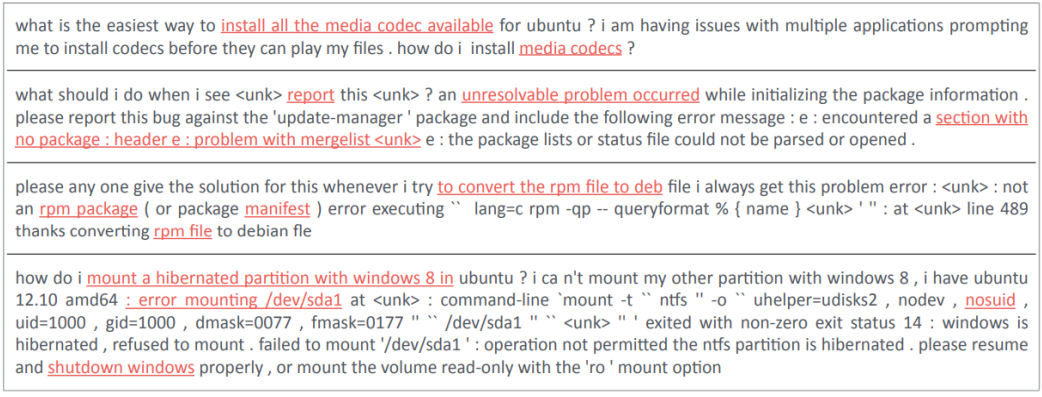


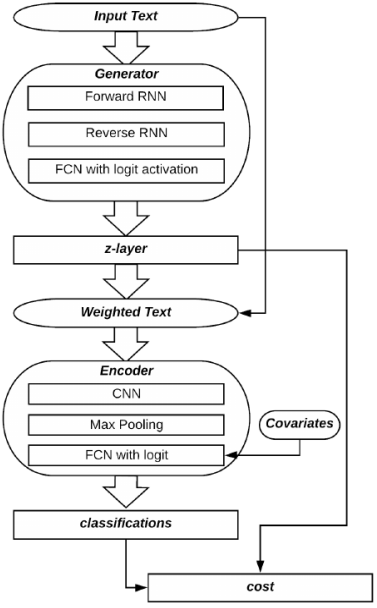
Figure 7: Examples of extracted rationales of questions in the AskUbuntu domain.

Outline

Discovering Influential Text

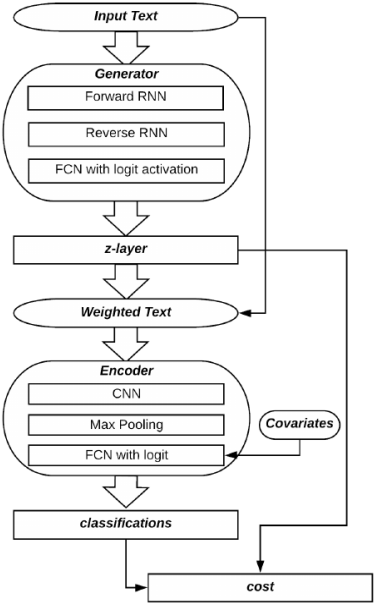
- Sanford, Roberts, and Li (2019) apply the Lei et al (2016) model in a social science setting.
- The stated goals:
  1. Recognize phrases of varying length that are predictive of an outcome.
  2. Allow for highlighted phrases to flexibly express the same idea.
  3. Map identified phrases to a common vector for interpretability.
  4. Allow for the inclusion of non-text covariates to aid in prediction.

Model Overview



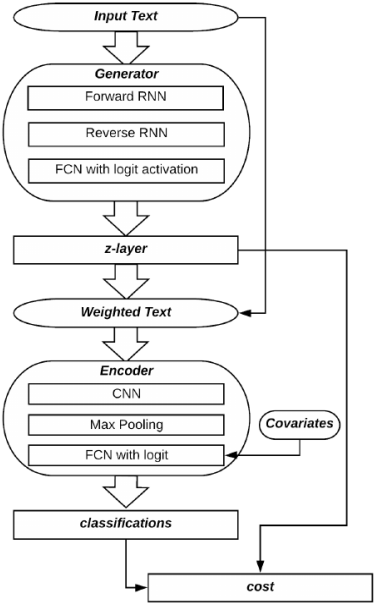
- Pre-Processing:
  - tokenize
  - convert to lower-case
  - pad to constant length (e.g. 100 words)

Model Overview



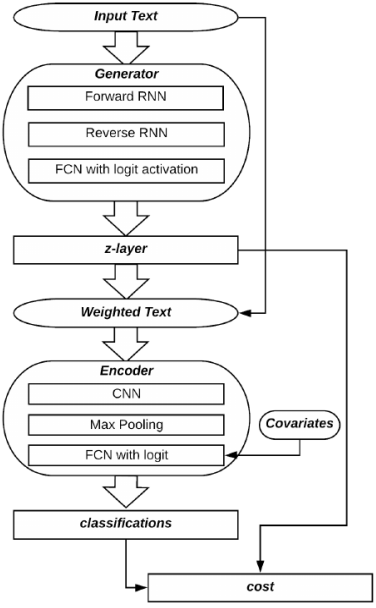
- Pre-Processing:
  - tokenize
  - convert to lower-case
  - pad to constant length (e.g. 100 words)
- Embedding:
  - 300-dimensional word embeddings, pre-initialized from fastText embeddings.

Model Overview



- Pre-Processing:
  - tokenize
  - convert to lower-case
  - pad to constant length (e.g. 100 words)
- Embedding:
  - 300-dimensional word embeddings, pre-initialized from fastText embeddings.
- Generator:
  1. $\boldsymbol{x} \rightarrow$ forward RNN $\rightarrow \boldsymbol{h_1}$
  2. $\boldsymbol{h_1} \rightarrow$ backward RNN $\rightarrow \boldsymbol{h_2}$
  3. $\boldsymbol{h_2} \rightarrow$ MLP $\rightarrow \boldsymbol{z}$, the rationale filters

Model Overview



- Pre-Processing:
  - tokenize
  - convert to lower-case
  - pad to constant length (e.g. 100 words)
- Embedding:
  - 300-dimensional word embeddings, pre-initialized from fastText embeddings.
- Generator:
  1. $\boldsymbol{x} \rightarrow$ forward RNN $\rightarrow \boldsymbol{h_1}$
  2. $\boldsymbol{h_1} \rightarrow$ backward RNN $\rightarrow \boldsymbol{h_2}$
  3. $\boldsymbol{h_2} \rightarrow$ MLP $\rightarrow \boldsymbol{z}$, the rationale filters
- Encoder
  - $\boldsymbol{x} \cdot \boldsymbol{z} \rightarrow \boldsymbol{w}$, the rationales
  - $\boldsymbol{w} \rightarrow$ convolution / pooling $\rightarrow \boldsymbol{c}$, feature map
  - $\boldsymbol{c} \rightarrow$ softmax (add covariates) $\rightarrow \hat{\boldsymbol{y}}$

Joint Objective

$$\underbrace{\mathcal{L}(z, x, y)}_{\text{cross-entropy}} + \lambda_1 \underbrace{(\sum_{i=1}^{n_z} z_i - \bar{z})^2}_{\text{sparsity}} + \lambda_2 \underbrace{\sum_{i=1}^{n_z} |z_i - z_{i-1}|}_{\text{coherence}}$$

- **cross-entropy**: rewards accurate predictions.
- **sparsity**: penalty on more rationales.
  - n_z= maximum rationale length.
  - $\bar{z}$ is a sparsity "target"; encourages rationales to be this length.
- **coherence**:
  - encourages high weights to be consecutive
  - encourages n-grams rather than single words.

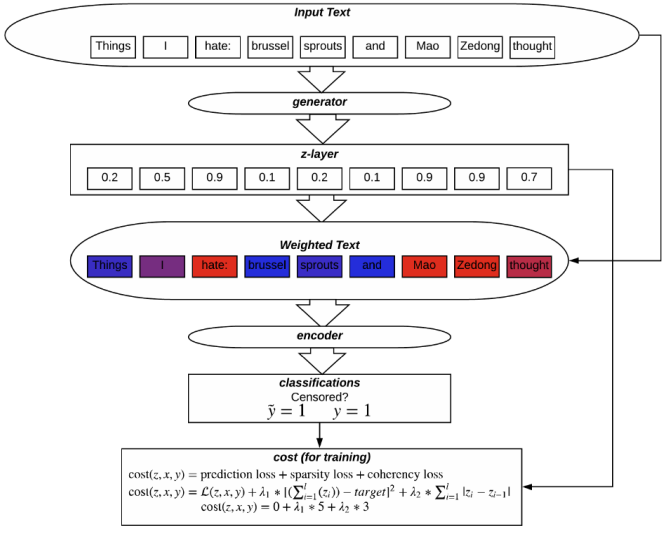


Figure 4: Detail view of how a text passes through the network. The z-layer tells the network which words to select, the rationale is a new text that the encoder uses to generate its classification. The loss function is composed of prediction error, a sparsity loss, and a coherency cost.

Interpreting the Rationales

- Model provides convolutional filters:
  - by themselves, not interpretable
  - but can see which (frequent) phrases in the corpus activate them.

Interpreting the Rationales

- Model provides convolutional filters:
  - by themselves, not interpretable
  - but can see which (frequent) phrases in the corpus activate them.
- Sanford et al find that doing this in the test set produces more intuitive rationales.

Example Rationales: Chinese Censorship

Phrase	Translation
饥荒 饿死 3600万	Famine that starved 36 million
被 暴徒 殴打	Beaten by a mob
甘肃 省委 常委	Gansu Provincial Party Committee Standing Committee
爱 国 主义 信仰	Patriotism
抢 劫 国有 资产	Robbery of state-owned assets
市委 书记 一一	Party secretary
变相 颠覆 国家	Disguised subversion
劳动 人民 在	Working People
人民 谋 和平	People want peace
人民 书记 人民	People's secretary, people
习近平 书记 领导	Xi Jinping Secretary leader
中央 经济 会议	Central economic conference
中国 民主 前途	China's democratic future

Table 1: Strongest Filter Phrases Associated With Censorship

Phrase	Translation
11 点 开 抢	Opens at 11
! !	!!
! ! 吧 !	!! right? !
! love your [爱	love your love
! your is so good	! your is so good
11 点 开	Opens at 11
>> http :	>> http :
! !	!!
美[爱	beautiful love
! !	!!
bb < 3	bb < 3
with bb < 3	with bb < 3
包邮 】 11 点	Shipping 11 o'clock
! ! 吧	right?

Table 2: Phrases Associated with Lowest Censorship

Corpus of 20,000 social media posts; 10,000 were censored; pre-trained fastText embeddings; 20 filters of width 3; 80% test-set accuracy.

Outline

Attention Mechanisms

- In 2014 the addition of "attention mechanisms" revolutionized neural machine translation.
  - and by 2019, NLP generally.
- The key difference is that a sequence is not represented as a single vector:
  - instead, produce a weighted aggregation of the sequence, filtered by their importance to a task.

Attention Filters

- Produce (embedding) vectors for the whole sentence.
  - e.g., use an RNN, e.g. bidirectional GRU, keeping the output vectors at each index.

$$\boldsymbol{x}_{1:n} = \text{RNN}(\boldsymbol{w}_{1:n})$$

  - but could just line up GloVe embeddings of each word, for example.

Attention Filters

- Produce (embedding) vectors for the whole sentence.
  - e.g., use an RNN, e.g. bidirectional GRU, keeping the output vectors at each index.

$$\boldsymbol{x}_{1:n} = \text{RNN}(\boldsymbol{w}_{1:n})$$

  - but could just line up GloVe embeddings of each word, for example.

- At the prediction stage (e.g. decoding), use an **attention filter** α_l to aggregate the context vectors:

$$\boldsymbol{h} = \sum_{l=1}^{n} \boldsymbol{\alpha}_l \cdot \boldsymbol{x}_l$$

Attention Filters

- Produce (embedding) vectors for the whole sentence.
  - e.g., use an RNN, e.g. bidirectional GRU, keeping the output vectors at each index.

$$\boldsymbol{x}_{1:n} = \mathrm{RNN}(\boldsymbol{w}_{1:n})$$

  - but could just line up GloVe embeddings of each word, for example.

- At the prediction stage (e.g. decoding), use an **attention filter** α_l to aggregate the context vectors:

$$\boldsymbol{h} = \sum_{l=1}^{n} \boldsymbol{\alpha}_l \cdot \boldsymbol{x}_l$$

- we have

$$\boldsymbol{\alpha}_l = \boldsymbol{\alpha}(\boldsymbol{x}_l, \boldsymbol{s})$$

  is a learnable MLP with softmax activation.
  - $\boldsymbol{s}$ represents the current task – e.g., the current decoder state for translation.

Attention Filters

- Produce (embedding) vectors for the whole sentence.
  - e.g., use an RNN, e.g. bidirectional GRU, keeping the output vectors at each index.

$$\boldsymbol{x}_{1:n} = \mathrm{RNN}(\boldsymbol{w}_{1:n})$$

  - but could just line up GloVe embeddings of each word, for example.

- At the prediction stage (e.g. decoding), use an **attention filter** $\boldsymbol{\alpha}_l$ to aggregate the context vectors:

$$\boldsymbol{h} = \sum_{l=1}^{n} \boldsymbol{\alpha}_l \cdot \boldsymbol{x}_l$$

- we have

$$\boldsymbol{\alpha}_l = \boldsymbol{\alpha}(\boldsymbol{x}_l, \boldsymbol{s})$$

  is a learnable MLP with softmax activation.
  - $\boldsymbol{s}$ represents the current task – e.g., the current decoder state for translation.
- Attention produces a weighted average of the context vectors ($\boldsymbol{\alpha}_l \in [0,1]^{n_\alpha}$).
- Uses $\boldsymbol{x}_{1:n}$ and $\boldsymbol{s}$ to extract most important information for task, e.g.:

$$\boldsymbol{y} = \mathrm{MLP}(\boldsymbol{h}, \boldsymbol{s})$$

Attention Filters are Interpretable

- Attention filter weights at each step when predicting the next word in the sentence:

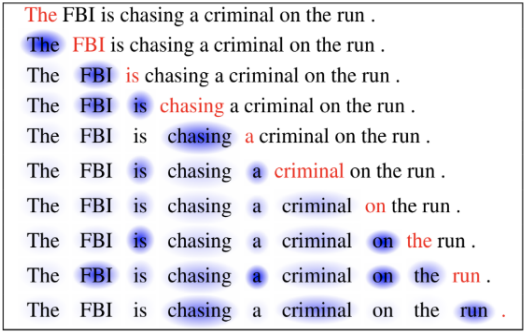


Fig. 6. The current word is in red and the size of the blue shade indicates the activation level. (Image source: Cheng et al., 2016)

Attention Filters are Interpretable

Figure 16-7. Visual attention: an input image (left) and the model's focus before producing the word "frisbee" (right)[18]

Self-Attention

- Say we have a sentence with words indexed by i or j.
- Input vectors $\boldsymbol{x}_{1:n}$ and output vectors $\boldsymbol{h}_{1:n}$ with

$$\boldsymbol{h}_i = \sum_{j=1}^{n} \boldsymbol{\alpha}_{ij} \cdot \boldsymbol{x}_j$$

Self-Attention

- Say we have a sentence with words indexed by i or j.
- Input vectors $\boldsymbol{x}_{1:n}$ and output vectors $\boldsymbol{h}_{1:n}$ with

$$\boldsymbol{h}_i = \sum_{j=1}^{n} \boldsymbol{\alpha}_{ij} \cdot \boldsymbol{x}_j$$

- The attention weight $\boldsymbol{\alpha}_{ij}$ is a function of $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$:

$$\boldsymbol{\alpha}_{ij} = \boldsymbol{\alpha}(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

Self-Attention

- Say we have a sentence with words indexed by i or j.
- Input vectors $\boldsymbol{x}_{1:n}$ and output vectors $\boldsymbol{h}_{1:n}$ with

$$\boldsymbol{h}_i = \sum_{j=1}^{n} \boldsymbol{\alpha}_{ij} \cdot \boldsymbol{x}_j$$

- The attention weight $\boldsymbol{\alpha}_{ij}$ is a function of $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$:

$$\boldsymbol{\alpha}_{ij} = \boldsymbol{\alpha}(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

- Specified as the logit-normalized dot product between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$:

$$\boldsymbol{\alpha}(\boldsymbol{x}_i, \boldsymbol{x}_j; \boldsymbol{x}_{1:n}) = \frac{\exp(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)}{\sum_k \exp(\boldsymbol{x}_i \cdot \boldsymbol{x}_k)}$$

Why self-attention works

- Consider a sentence

  $$\text{the, cat, walks, on, the, street}$$

  with embeddings

  $$\boldsymbol{x}_{\text{the}}, \boldsymbol{x}_{\text{cat}}, \boldsymbol{x}_{\text{walks}}, \boldsymbol{x}_{\text{on}}, \boldsymbol{x}_{\text{the}}, \boldsymbol{x}_{\text{street}}$$

- Feeding this sentence into the self-attention layer produces

  $$\boldsymbol{h}_{\text{the}}, \boldsymbol{h}_{\text{cat}}, \boldsymbol{h}_{\text{walks}}, \boldsymbol{h}_{\text{on}}, \boldsymbol{h}_{\text{the}}, \boldsymbol{h}_{\text{street}}$$

Why self-attention works

- Consider a sentence

  $$\text{the, cat, walks, on, the, street}$$

  with embeddings

  $$\boldsymbol{x}_{\text{the}}, \boldsymbol{x}_{\text{cat}}, \boldsymbol{x}_{\text{walks}}, \boldsymbol{x}_{\text{on}}, \boldsymbol{x}_{\text{the}}, \boldsymbol{x}_{\text{street}}$$

- Feeding this sentence into the self-attention layer produces

  $$\boldsymbol{h}_{\text{the}}, \boldsymbol{h}_{\text{cat}}, \boldsymbol{h}_{\text{walks}}, \boldsymbol{h}_{\text{on}}, \boldsymbol{h}_{\text{the}}, \boldsymbol{h}_{\text{street}}$$

- Where we have

  $$\boldsymbol{h}_i = \sum_{j=1}^{n} \frac{\exp(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)}{\sum_k \exp(\boldsymbol{x}_i \cdot \boldsymbol{x}_k)} \cdot \boldsymbol{x}_j$$

  the weighted average of the embedding vectors of the other words j in the sentence, weighted by the logit-normalized dot product with $\boldsymbol{x}_i$.

Why self-attention works

- Consider a sentence

  the, cat, walks, on, the, street

  with embeddings

  $$\boldsymbol{x}_{\text{the}}, \boldsymbol{x}_{\text{cat}}, \boldsymbol{x}_{\text{walks}}, \boldsymbol{x}_{\text{on}}, \boldsymbol{x}_{\text{the}}, \boldsymbol{x}_{\text{street}}$$

- Feeding this sentence into the self-attention layer produces

  $$\boldsymbol{h}_{\text{the}}, \boldsymbol{h}_{\text{cat}}, \boldsymbol{h}_{\text{walks}}, \boldsymbol{h}_{\text{on}}, \boldsymbol{h}_{\text{the}}, \boldsymbol{h}_{\text{street}}$$

- Where we have

  $$\boldsymbol{h}_i = \sum_{j=1}^{n} \frac{\exp(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)}{\sum_k \exp(\boldsymbol{x}_i \cdot \boldsymbol{x}_k)} \cdot \boldsymbol{x}_j$$

  the weighted average of the embedding vectors of the other words j in the sentence, weighted by the logit-normalized dot product with $\boldsymbol{x}_i$.

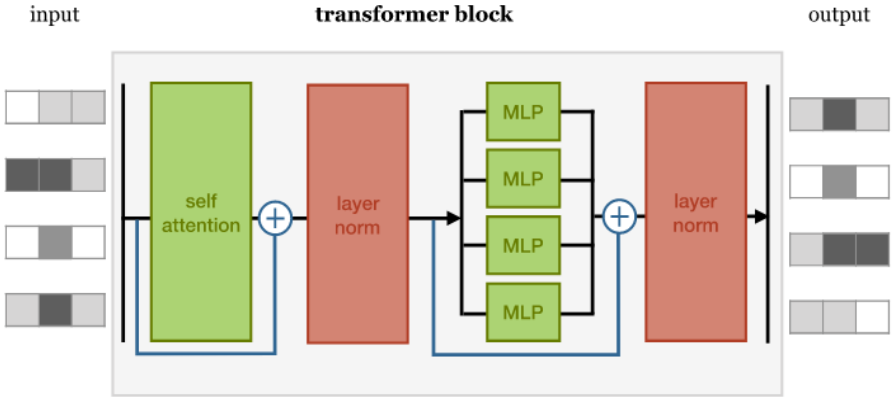
- The embeddings $\boldsymbol{x}$ are learned for a task:
  - For example, in many tasks (e.g. predicting the next sentence), stopwords like "the" will not be helpful.
  - Therefore the embedding $\boldsymbol{x}_{\text{the}}$ will end up having a low or negative dot product with more informative words.

Attention is All you Need: Transformers

- Since a 2017 paper (Vaswani et al 2017), deep learning for NLP has been transformed by a new class of models:
  - **transformers** fully replace recurrence or convolutions with attention mechanisms.

Attention is All you Need: Transformers

- Since a 2017 paper (Vaswani et al 2017), deep learning for NLP has been transformed by a new class of models:
  - **transformers** fully replace recurrence or convolutions with attention mechanisms.
- The transformer building block:



Technicalities

Multi-Head Attention:

- in a single self-attention layer, multiple self-attention mechanisms operate in parallel
- allows specialization across functions, e.g. verb-subject vs verb-object

Technicalities

Multi-Head Attention:

- in a single self-attention layer, multiple self-attention mechanisms operate in parallel
- allows specialization across functions, e.g. verb-subject vs verb-object

Query-Key-Value Weights:

- see sources on syllabus

Technicalities

Multi-Head Attention:

- in a single self-attention layer, multiple self-attention mechanisms operate in parallel
- allows specialization across functions, e.g. verb-subject vs verb-object

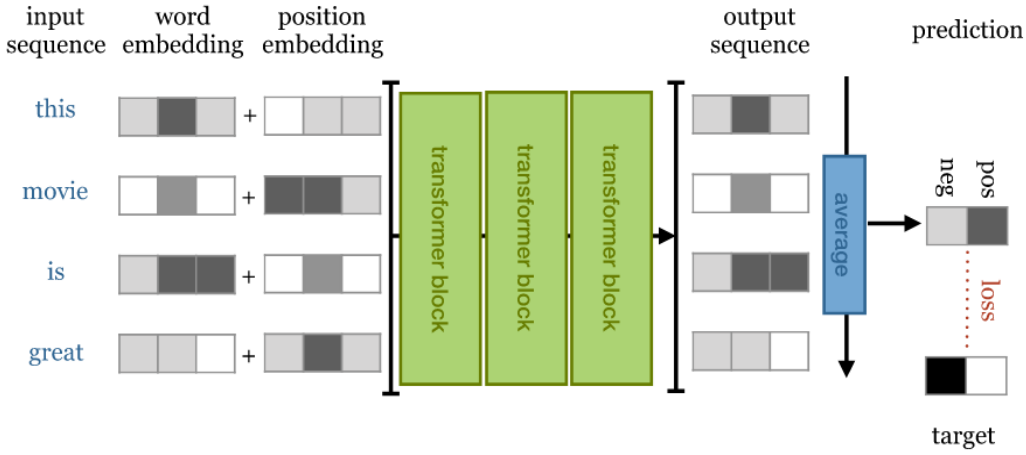
Query-Key-Value Weights:

- see sources on syllabus

Positional Embeddings:

- can add learnable categorical embeddings for the position (index) of the word in the sentence / document.
  - puts a hard limit on sequence lengths.
- or can add a set of non-linear transformations of the index, e.g. sine / cosine.

Transformer for Sentiment Classification



- will get state-of-the-art performance, but faster to train than a bidirectional LSTM.