Language Models for Law and Social Science

7. Sequence Embeddings

Social Science with Embeddings

Social Science with Embeddings

- What are we measuring?
  - e.g., how do measurements by person or by group correlate with actual attitudes, for example as measured by IAT scores?

Social Science with Embeddings

- What are we measuring?
  - e.g., how do measurements by person or by group correlate with actual attitudes, for example as measured by IAT scores?
- What attitudes can be reliably measured?
  - e.g. racial sentiment, especially in light of black sheep problem.

Social Science with Embeddings

- What are we measuring?
  - e.g., how do measurements by person or by group correlate with actual attitudes, for example as measured by IAT scores?
- What attitudes can be reliably measured?
  - e.g. racial sentiment, especially in light of black sheep problem.
- In what domains is this relevant?
  - social media, news media, politics, legal, scientific, ...

Social Science with Embeddings

- What are we measuring?
  - e.g., how do measurements by person or by group correlate with actual attitudes, for example as measured by IAT scores?
- What attitudes can be reliably measured?
  - e.g. racial sentiment, especially in light of black sheep problem.
- In what domains is this relevant?
  - social media, news media, politics, legal, scientific, ...
- Does language matter?
  - Djourelova (2020): style change from “illegal” to “undocumented” immigrant softened attitudes toward immigration.

Examples: Confounders in Measurement from Text

What quantity do we care about? vs. What do we measure?

Examples: Confounders in Measurement from Text

What quantity do we care about? vs. What do we measure?

- Positive/negative sentiment $\rightarrow$ Count positive/negative words, or predict text annotations.
- Toxicity $\rightarrow$ Count toxic words, or predict text annotations.

confounders?

Examples: Confounders in Measurement from Text

What quantity do we care about? vs. What do we measure?

- Positive/negative sentiment → Count positive/negative words, or predict text annotations.
- Toxicity → Count toxic words, or predict text annotations.

confounders?

- Student performance → predicted essay grade based on labeled essay documents.
- Credit worthiness → predicted probability of default based on loan application documents.

confounders?

Examples: Confounders in Measurement from Text

What quantity do we care about? vs. What do we measure?

- Positive/negative sentiment $\rightarrow$ Count positive/negative words, or predict text annotations.
- Toxicity $\rightarrow$ Count toxic words, or predict text annotations.

confounders?

- Student performance $\rightarrow$ predicted essay grade based on labeled essay documents.
- Credit worthiness $\rightarrow$ predicted probability of default based on loan application documents.

confounders?

- Political partisanship $\rightarrow$ predicted probability being Democrat/Republican based on speeches.

confounders?

Examples: Confounders in Measurement from Text

What quantity do we care about? vs. What do we measure?

- Positive/negative sentiment $\rightarrow$ Count positive/negative words, or predict text annotations.
- Toxicity $\rightarrow$ Count toxic words, or predict text annotations.

confounders?

- Student performance $\rightarrow$ predicted essay grade based on labeled essay documents.
- Credit worthiness $\rightarrow$ predicted probability of default based on loan application documents.

confounders?

- Political partisanship $\rightarrow$ predicted probability being Democrat/Republican based on speeches.

confounders?

- Policy priorities $\rightarrow$ predicted probability of speeches/laws being about a particular policy topic.

confounders?

When is measurement confounding important?

- By itself, producing measurements that are biased by confounders might not be a problem.
- e.g.:
  - an NLP-based essay grading system that learns confounders $\rightarrow$ not a problem unless tudents learn about it and strategically alter their essays.

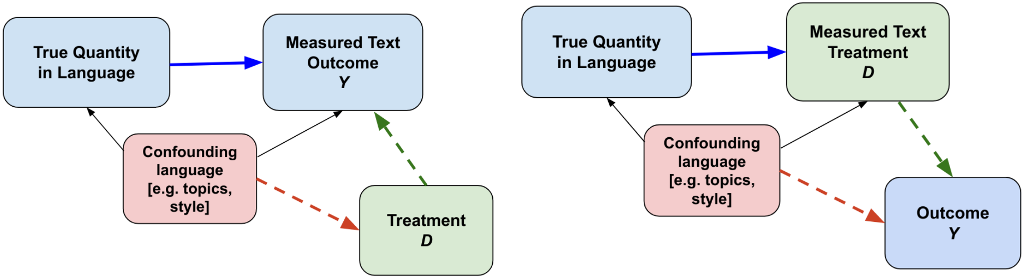
When is measurement confounding important?

- By itself, producing measurements that are biased by confounders might not be a problem.
- e.g.:
  - an NLP-based essay grading system that learns confounders $\rightarrow$ not a problem unless tudents learn about it and strategically alter their essays.
- for measuring political divisiveness or policy priorities
  - probably won't matter for in-domain summary statistics
  - but would matter a lot for summary statistics in a new domain

When is measurement confounding important?

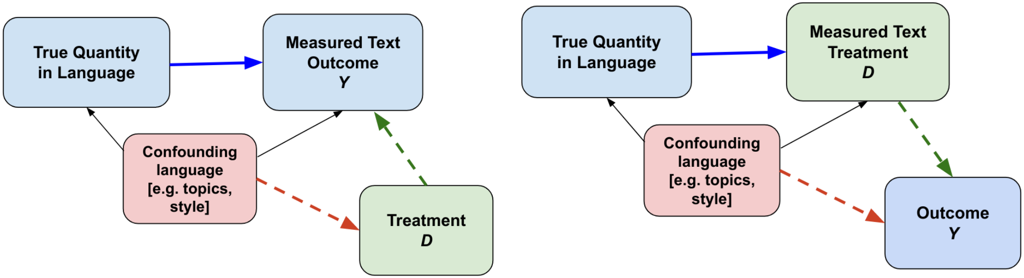
- By itself, producing measurements that are biased by confounders might not be a problem.
- e.g.:
  - an NLP-based essay grading system that learns confounders → not a problem unless tudents learn about it and strategically alter their essays.
- for measuring political divisiveness or policy priorities
  - probably won't matter for in-domain summary statistics
  - but would matter a lot for summary statistics in a new domain
- even in-domain, will matter for assessing the causal effect of a treatment, e.g. the electoral cycle:
  - elections might cause politicians to focus on social issues rather than economic issues,
  - if social/economic issues are confounded with partisanship, the resulting estimates are biased.

When is measurement confounding important?



- When text is outcome, the confounders cannot be correlated with the treatment.
- When text is treatment, the confounders cannot be correlated with the outcome.

When is measurement confounding important?



- When text is outcome, the confounders cannot be correlated with the treatment.
- When text is treatment, the confounders cannot be correlated with the outcome.
  - e.g.: estimating the effect of politician speech sentiment on his/her reelection chances?

Next week: Online Text-as-data workshop

- Schedule and zoom linked on syllabus
  - Monday: 8am-11am, 5pm-8pm
  - Tuesday: 8am-11am
- Extra credit (1 point on response essay):
  - Watch at least 2 presentations (20 minutes each) and ask at least 1 question.
  - Screenshot it and send it to jingwei.

What is (Document) Embedding?

“Embedding”: a lower-dimensional dense vector representation of a higher-dimensional object

- also refers to algorithm for making such vectors

What is (Document) Embedding?

"Embedding": a lower-dimensional dense vector representation of a higher-dimensional object

- also refers to algorithm for making such vectors

Document vectors:

- quantitative analysis of language requires that documents be transformed to numbers – that is, vectors.

What is (Document) Embedding?

"Embedding": a lower-dimensional dense vector representation of a higher-dimensional object

- also refers to algorithm for making such vectors

Document vectors:

- quantitative analysis of language requires that documents be transformed to numbers – that is, vectors.
- standard approach:
  - represent documents as sparse vectors of token counts/frequencies.
  - e.g., to compute document similarity measures, to learn topics, or to use as features in supervised learning

What is (Document) Embedding?

"Embedding": a lower-dimensional dense vector representation of a higher-dimensional object

- also refers to algorithm for making such vectors

Document vectors:

- quantitative analysis of language requires that documents be transformed to numbers – that is, vectors.
- standard approach:
  - represent documents as sparse vectors of token counts/frequencies.
  - e.g., to compute document similarity measures, to learn topics, or to use as features in supervised learning
- **Embedding approach**:
  - low-dimensional dense vectors rather than high-dimensional sparse vectors
  - Embedding without neural nets:
    - PCA reductions of the document-term matrix
    - LDA topic shares

What is (Document) Embedding?

"Embedding": a lower-dimensional dense vector representation of a higher-dimensional object

- also refers to algorithm for making such vectors

Document vectors:

- quantitative analysis of language requires that documents be transformed to numbers – that is, vectors.
- standard approach:
  - represent documents as sparse vectors of token counts/frequencies.
  - e.g., to compute document similarity measures, to learn topics, or to use as features in supervised learning
- **Embedding approach**:
  - low-dimensional dense vectors rather than high-dimensional sparse vectors
  - Embedding without neural nets:
    - PCA reductions of the document-term matrix
    - LDA topic shares
  - Embedding with neural nets (today):
    - many useful ways to do this.

Outline

Word Vectors can produce Document Vectors

$$\vec{D} = \sum_{w \in D} a_w \vec{w}$$

- The "continuous bag of words" (CBOW) representation for document D is the sum, or the average (potentially weighted by a_w), of the vectors $\vec{w}$ for each word w in the document.
  - word vectors $\vec{w}$ constructed using pre-trained GloVe or Word2Vec.
  - "Document" could be sentence, paragraph, section, etc. (scales well to long docs)

Word Vectors can produce Document Vectors

$$\vec{D} = \sum_{w \in D} a_w \vec{w}$$

- The "continuous bag of words" (CBOW) representation for document D is the sum, or the average (potentially weighted by a_w), of the vectors $\vec{w}$ for each word w in the document.
  - word vectors $\vec{w}$ constructed using pre-trained GloVe or Word2Vec.
  - "Document" could be sentence, paragraph, section, etc. (scales well to long docs)
- Arora, Liang, and Ma (2017) provide a "tough to beat baseline", the SIF-weighted ("smoothed inverse frequency") average of the vectors:

$$a_w = \frac{\alpha}{\alpha + p_w}$$

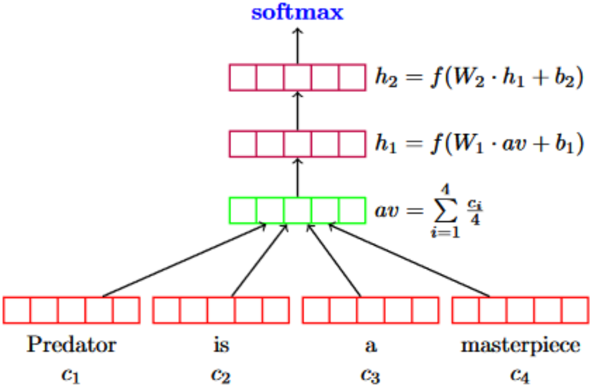
  where p_w is the probability (frequency) of the word and $\alpha = .001$ is a smoothing parameter.

Deep Averaging Network (Iyyer et al 2015)

- Similar to averaged word embeddings across doc, but embeddings are learned during neural net training:

Deep Averaging Network (Iyyer et al 2015)

- Similar to averaged word embeddings across doc, but embeddings are learned during neural net training:



1. Trainable embedding layer for words, initialized with pre-trained embeddings
2. Average the embeddings, with dropout (sometimes words left out of average)
3. Average embedding fed into MLP with multiple hidden layers
4. MLP outputs used for classification or regression

Hashed N-Gram Embeddings (Joulin et al 2016)

Combine the Iyyer et al (2015) approach with the hashing n-gram vectorizer.

Hashed N-Gram Embeddings (Joulin et al 2016)

Combine the Iyyer et al (2015) approach with the hashing n-gram vectorizer.

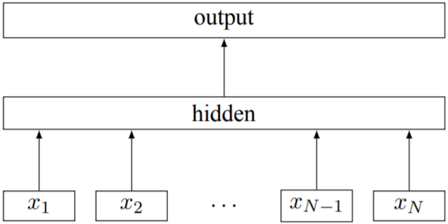


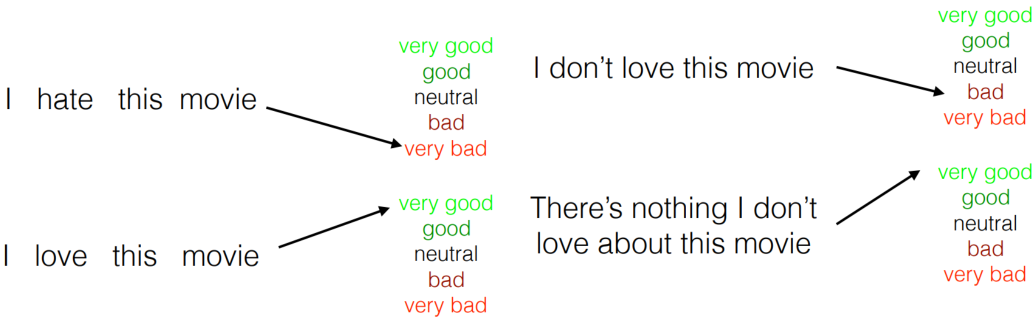
Figure 1: Model architecture of `fastText` for a sentence with N ngram features $x_1, \ldots, x_N$. The features are embedded and averaged to form the hidden variable.

1. Allocate $n_w \approx 10$ million rows to embedding matrix.
2. Assign n-grams to embedding indexes with hashing function.
3. sentence embedding $=$ average of n-gram embeddings
4. send to dense hidden layer(s)
5. send to output (e.g. classifier / regressor).

- Captures local word-order information from n-grams without building vocabulary or costly training of Convolutional Neural Net.

Outline

The Classic Sentence Classification Problem



Source: Graham Neubig slides.

- bag-of-words models won't capture the importance of "don't love" or "nothing I don't love", even with interactions / hidden layers.
- N-grams have a large feature space (especially with 4-grams) and don't share information across similar words/n-grams.

Sequence Data

- The real break-through from deep learning for NLP:
  - moving from bag-of-X representations to sequence representations.
  - Rather than inputting **counts over words/n-grams** $\boldsymbol{x}$, take as input a **sequence of tokens** $\{w_{1},...,w_t,...w_n\}$.

Sequence Data

- The real break-through from deep learning for NLP:
  - moving from bag-of-X representations to sequence representations.
  - Rather than inputting **counts over words/n-grams** $\boldsymbol{x}$, take as input a **sequence of tokens** $\{w_1, ..., w_t, ...w_n\}$.
- "Traditional" architectures:
  - Convolutional neural nets (CNNs)
  - Recurrent Neural Nets (RNNs)
- "Modern" architectures:
  - Transformers ("attentional" neural nets) and variants

From last time: Embedding every token in a document

- **Embedding layers** take a categorical variable as input and produce a low-dimensional dense representation.

From last time: Embedding every token in a document

- **Embedding layers** take a categorical variable as input and produce a low-dimensional dense representation.

We saw this example last time, which produces document embeddings:

- Tokenize document to fixed length n_L
- Inputs are each word position, input categorical (word) to n_E-dimensional embedding layer:

$$\boldsymbol{x}_{1:n_L} = [\ \boldsymbol{x}_1 \quad \dots \quad \boldsymbol{x}_t \quad \dots \quad \boldsymbol{x}_{n_L}\ ]$$

- pipe to further hidden layers of network.

From last time: Embedding every token in a document

- **Embedding layers** take a categorical variable as input and produce a low-dimensional dense representation.

We saw this example last time, which produces document embeddings:

- Tokenize document to fixed length n_L
- Inputs are each word position, input categorical (word) to n_E-dimensional embedding layer:

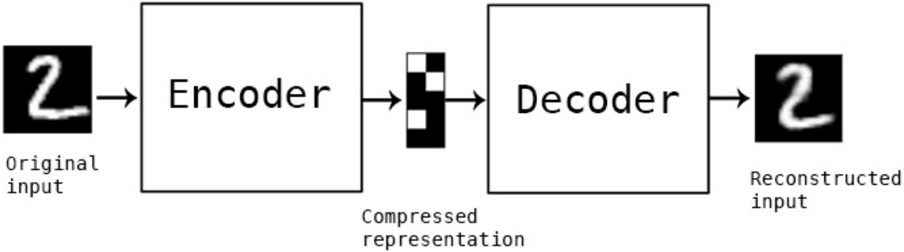
$$\boldsymbol{x}_{1:n_L} = [\ \boldsymbol{x}_1 \quad \ldots \quad \boldsymbol{x}_t \quad \ldots \quad \boldsymbol{x}_{n_L}\ ]$$

- pipe to further hidden layers of network.
- document embedding $= n_L n_E$-dimensional vector of concatenated word embeddings.
  - computationally demanding and only works with short documents.

Autoencoders: Optimal Compression Algorithms

Autoencoders: Optimal Compression Algorithms

- Autoencoders = neural nets that perform domain-specific lossy compression:



- Learned encodings can be decoded back to a *reconstruction* – a (minimally) lossy representation of the original data.
- AE's can memorize complex, unstructured data – deep unsupervised learning.

Autoencoder Architecture – Neural net with output=input

- Stacked layers gradually decrease in dimensionality to create the compressed representation
- then gradually increase in dimensionality to try to reconstruct the input.

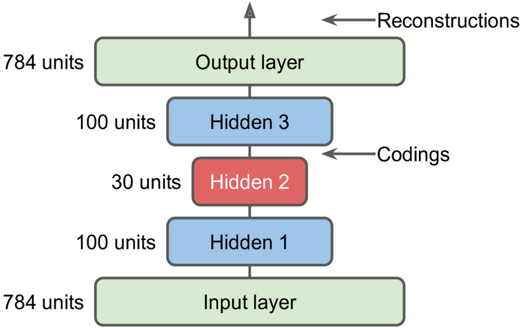


Figure 17-3. Stacked autoencoder

Reconstruction from encoded vector

Figure 17-4. Original images (top) and their reconstructions (bottom)

Autoencoder Encodings are Embeddings

- Autoencoder compresses a document (e.g. a sentence) into a vector to be reconstructed.
  - Can use the compressed representation as a document embedding.
- Standard (that is, non-transformer) autoencoder embeddings don't tend to work well for sentence similarity tasks because autoencoders try to reproduce the specific wording (reconstruction objective), rather than the semantic meaning.
  - transformer-based autoencoders, i.e. BART, address this issue (next week)

Convolutional Neural Nets ↔ N-gram Detectors

- A neural net architecture that constructs **filters** that slide across input sequences and extract **local predictive structure**.

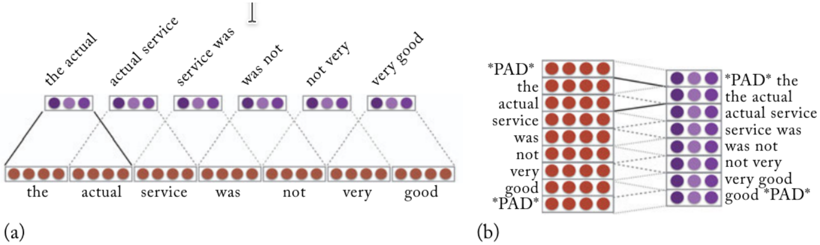


Figure 13.1: The inputs and outputs of a narrow and a wide convolution in the vector-concatenation and the vector-stacking notations. (a) A *narrow* convolution with a window of size $k = 2$ and 3-dimensional output ($\ell = 3$), in the vector-concatenation notation. (b) A *wide* convolution with a window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

- Overall, CNNs do not work well in NLP; use embedded hashed n-grams instead (Joulin et al 2016, Goldberg 2017).

RNNs can input and output arbitrary-length sequences

- Downsides of previous approaches:
  - CBOW models (averaged word/phrase embeddings) lose any sequence information beyond local word order encoded by n-grams
  - all-token embedding, and CNNs, require fixed-length documents

RNNs can input and output arbitrary-length sequences

- Downsides of previous approaches:
  - CBOW models (averaged word/phrase embeddings) lose any sequence information beyond local word order encoded by n-grams
  - all-token embedding, and CNNs, require fixed-length documents
- Recurrent Neural Nets (RNNs) work with **sequences of arbitrary length**, both as **inputs** and **outputs**:
  - can *encode* sequences into vectors.
  - can *decode* vectors into sequences.
- therefore especially useful for language tasks such as translation.

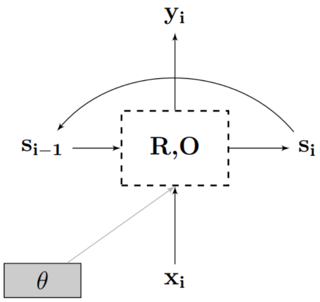
Summary of RNN Architecture

- At each step t:
  - a recursion function $R(s_{t-1}, x_t; \theta_R)$ computes the state vector s_t given current word x_t and previous state s_{t-1}.

Summary of RNN Architecture

- At each step t:
  - a recursion function $R(s_{t-1}, x_t; \theta_R)$ computes the state vector s_t given current word x_t and previous state s_{t-1}.
  - An output function $O(s_t; \theta_O)$ computes an output vector y_t (to be compared to the outcome variable in the dataset).

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



- The parameters of those functions, $\theta = (\theta_R, \theta_O)$ are learned during model training.

RNN Encoding and Decoding

top left: sequence to sequence; top right: sequence to vector

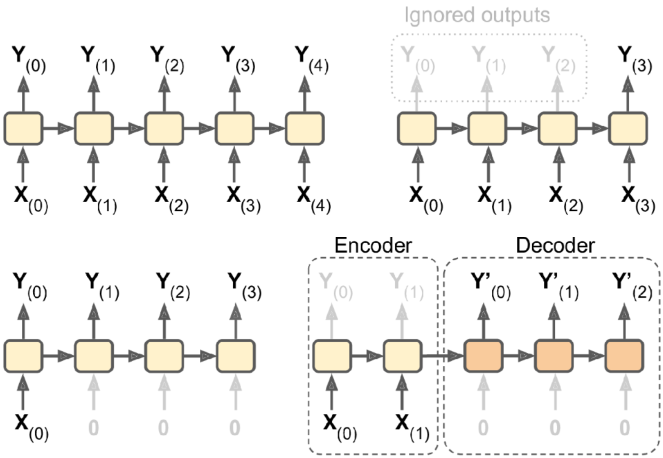


Figure 15-4. Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and Encoder–Decoder (bottom right) networks

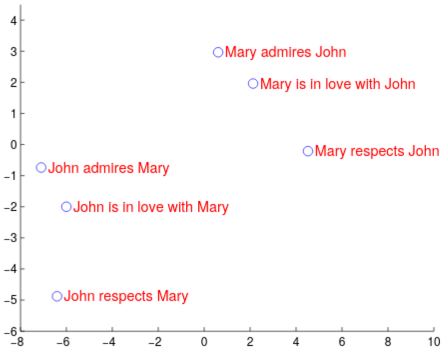
bottom left: vector to sequence; bottom right: encoder-decoder.

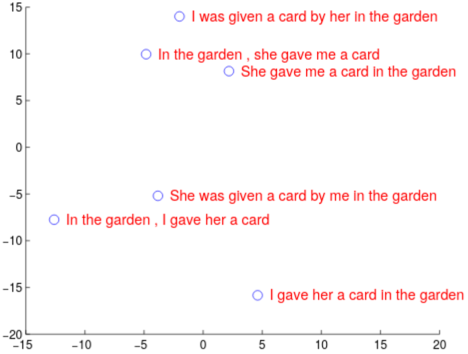
Gated RNNs – LSTM (Long Short-Term Memory)

- Gating mechanisms prevent vanishing/exploding gradients.
- bidirectional LSTMs (trained backward and forward) get state-of-the-art performance on text classification of short documents (e.g. classifying sentences by sentiment), but rarely better than transformer models.
- See Goldberg (2017) if curious.

RNN's (e.g. Machine Translation) Produce Document Embeddings

- RNN machine translators produce a sentence vector that must be decoded into another language.
- if the vector produces a good translation, it must contain the important information in the sentence.





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

Outline

Deep Learning with NLP $\approx$ Transformers

- Since a 2017 paper (Vaswani et al 2017), most deep learning for NLP uses the transformer architecture.
- Recurrent neural nets can process whole documents word-by-word, but they have to sweep through the whole document at each training epoch. They learn too slowly.
- Transformers overcome this limitation:
  - intuitively, they provide a way to efficiently read in an entire document and learn the meaning of all words and all interactions between words.

Transformers = Attentional Neural Nets

- Attention refers to a neural-net architecture/layer, like dense layers, embedding layers, convolutions, or recurrence mechanisms.

Transformers = Attentional Neural Nets

- Attention refers to a neural-net architecture/layer, like dense layers, embedding layers, convolutions, or recurrence mechanisms.

Attention does two things:

1. On a technical level:
    - allows a neural net to build many implicit databases of key-value pairs (a la python dictionaries), and to efficiently query those databases.

Transformers = Attentional Neural Nets

- Attention refers to a neural-net architecture/layer, like dense layers, embedding layers, convolutions, or recurrence mechanisms.

Attention does two things:

1. On a technical level:
   - allows a neural net to build many implicit databases of key-value pairs (a la python dictionaries), and to efficiently query those databases.

2. On a linguistic level:
   - allows a neural net to build a set of implicit key-value databases:
     - the keys are pairs of words
     - the value is a learnable vector that helps in some prediction task, e.g. predicting the next word in a sequence.

Attention heads

- Transformers consist of stacked blocks of parallel **attention heads**
- **Attention heads** are machine-reading filters, which allow each word to scan over every other word in the document and pick up predictive interactions.

GPT and BERT are Pre-Trained Transformer Models

- **GPT = "Generative Pre-Trained Transformer"**:
  - train transformer to predict the next word at the end of a sequence.
  - Three versions (GPT-2, GPT-2, GPT-3)
    - newer versions (GPT-3.5, GPT-4) use reinforcement learning with human alignment

GPT and BERT are Pre-Trained Transformer Models

- **GPT = "Generative Pre-Trained Transformer"**:
  - train transformer to predict the next word at the end of a sequence.
  - Three versions (GPT-2, GPT-2, GPT-3)
    - newer versions (GPT-3.5, GPT-4) use reinforcement learning with human alignment
  - notably good for text generation (language modeling)

GPT and BERT are Pre-Trained Transformer Models

- **GPT = "Generative Pre-Trained Transformer"**:
  - train transformer to predict the next word at the end of a sequence.
  - Three versions (GPT-2, GPT-2, GPT-3)
    - newer versions (GPT-3.5, GPT-4) use reinforcement learning with human alignment
  - notably good for text generation (language modeling)
- **BERT = "Bidirectional Encoder Representations from Transormers"**:
  - train transformer to predict left-out words in the middle of a sequence.

GPT and BERT are Pre-Trained Transformer Models

- **GPT = "Generative Pre-Trained Transformer"**:
  - train transformer to predict the next word at the end of a sequence.
  - Three versions (GPT-2, GPT-2, GPT-3)
    - newer versions (GPT-3.5, GPT-4) use reinforcement learning with human alignment
  - notably good for text generation (language modeling)
- **BERT = "Bidirectional Encoder Representations from Transormers"**:
  - train transformer to predict left-out words in the middle of a sequence.
  - the resulting document embeddings contain most (perhaps all?) of the relevant information in short language snippets.
    - blew away all the NLP baselines (e.g. semantic role labeling, question-answering, entailment, etc.) when it came out in 2018.

GPT and BERT are Pre-Trained Transformer Models

- **GPT = "Generative Pre-Trained Transformer"**:
  - train transformer to predict the next word at the end of a sequence.
  - Three versions (GPT-2, GPT-2, GPT-3)
    - newer versions (GPT-3.5, GPT-4) use reinforcement learning with human alignment
  - notably good for text generation (language modeling)
- **BERT = "Bidrectional Encoder Representations from Transormers"**:
  - train transformer to predict left-out words in the middle of a sequence.
  - the resulting document embeddings contain most (perhaps all?) of the relevant information in short language snippets.
    - blew away all the NLP baselines (e.g. semantic role labeling, question-answering, entailment, etc.) when it came out in 2018.
- immediately relevant use cases for our purpose:
  - many pre-trained models, e.g. for sentiment classification
  - BERT model can be fine-tuned to quickly get optimal results for many text classification tasks.

Shortcut: Using huggingface Pre-Trained Models

Shortcut: Using huggingface Pre-Trained Models

```python
from transformers import pipeline
sentiment_analysis = pipeline(“sentiment-analysis”)

pos_text = "I enjoy studying computational algorithms."
neg_text = "I dislike sleeping late everyday."

pos_sent = sentiment_analysis(pos_text)[0]
print(pos_sent['label'],0 pos_sent['score'])

neg_sent = sentiment_analysis(neg_text)[0]
print(neg_sent['label'], neg_sent['score'])
```

- also straightforward to fine-tune BERT for your own classification tasks.
- see notebooks for full details / explanation.

Queries, Keys, and Values

- Assume a database $D = \{(k_1, v_1), ...(k_m, v_m)\}$
  - m tuples of keys and values.
  - Denote by q a "query".
  - e.g., a python dictionary; query is used to look-up; normally one of the keys.

Queries, Keys, and Values

- Assume a database $D = \{(k_1, v_1), ...(k_m, v_m)\}$
  - m tuples of keys and values.
  - Denote by q a "query".
  - e.g., a python dictionary; query is used to look-up; normally one of the keys.

Define

$$\text{Attention}(q) = \sum_{i=1}^{m} a(q, k_i) v_i$$

- $a(\cdot)$ are scalar "attention weights"; they give more weight ("pay more attention") to some items based on q and k_i.
- In a normal database query / dictionary, $a(q, k) = 1$ if $q = k$ and zero otherwise.

Queries, Keys, and Values

- Assume a database $D = \{(k_1, v_1), ...(k_m, v_m)\}$
  - m tuples of keys and values.
  - Denote by q a "query".
  - e.g., a python dictionary; query is used to look-up; normally one of the keys.

Define

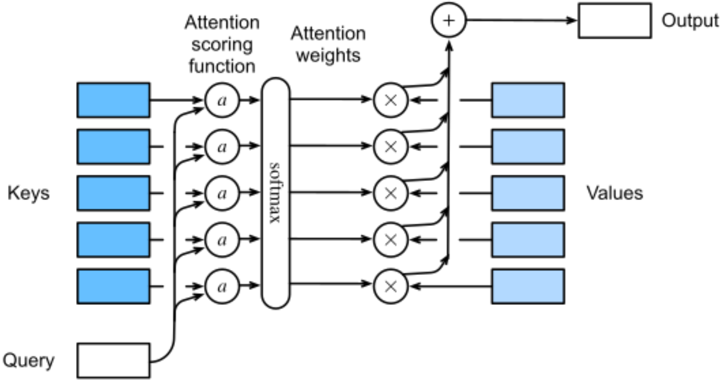
$$\text{Attention}(q) = \sum_{i=1}^{m} a(q, k_i) v_i$$

- $a(\cdot)$ are scalar "attention weights"; they give more weight ("pay more attention") to some items based on q and k_i.
- In a normal database query / dictionary, $a(q, k) = 1$ if $q = k$ and zero otherwise.
- in a transformer, this is generalized such that $a(\cdot) \geq 0$, $\sum a(\cdot) = 1$.
- achieved for any weighting function a_0 by a softmax operation:

$$a(q, k_i) = \text{softmax}(a_o(q, k_i)) = \frac{\exp(a_0(q, k_i))}{\sum_j \exp(a_o(q, k_j))}$$

  - $\uparrow$ differentiable and gradient never vanishes.

Scaled dot product attention



- let q and k be vectors with dimension d.
- scaled dot product attention:

$$a(q, k_i) = \text{softmax}(\frac{q \cdot k_i}{\sqrt{d}})$$

Self-Attention with word embeddings

- Consider a sequence of tokens with fixed length n_L, $\{w_1, ..., w_i, ..., w_{n_L}\}$
- We have (learnable) word embeddings $x_i = \omega_E w_i$ with dimension n_E, producing a sequence of vectors
$$\{x_1, ..., x_i, ..., x_{n_L}\}$$

Self-Attention with word embeddings

- Consider a sequence of tokens with fixed length n_L, $\{w_1, ..., w_i, ..., w_{n_L}\}$
- We have (learnable) word embeddings $x_i = \omega_E w_i$ with dimension n_E, producing a sequence of vectors

$$\{x_1, ..., x_i, ..., x_{n_L}\}$$

- A **self-attention layer** transforms $x_{1:n_L}$ into a second sequence $h_{1:n_L}$ with

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

  - where $a(\cdot)$ is an attention function such that $a(\cdot) \geq 0$, $\sum a(\cdot) = 1$.
  - $\rightarrow$ each h_i becomes a weighted average of the whole sequence.

Self-Attention with word embeddings

- Consider a sequence of tokens with fixed length n_L, $\{w_1, ..., w_i, ..., w_{n_L}\}$
- We have (learnable) word embeddings $x_i = \omega_E w_i$ with dimension n_E, producing a sequence of vectors

$$\{x_1, ..., x_i, ..., x_{n_L}\}$$

- A **self-attention layer** transforms $x_{1:n_L}$ into a second sequence $h_{1:n_L}$ with

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

  - where $a(\cdot)$ is an attention function such that $a(\cdot) \geq 0$, $\sum a(\cdot) = 1$.
  - $\rightarrow$ each h_i becomes a weighted average of the whole sequence.
  - if sequence length $n < n_L$, set $a_i = 0$ for all $i > n$.
- $h_{1:n_L}$ is flattened and piped to the network's hidden layers.

Basic Self-Attention

Setup:

1. Sequence of tokens $\{w_1, ..., w_i, ..., w_{n_L}\}$
2. Sequence of (trainable) embedding vectors $\{x_1, ..., x_i, ..., x_{n_L}\}$
3. Sequence of attention-transformed vectors $\{h_1, ..., h_i, ..., h_{n_L}\}$ with

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

Basic Self-Attention

Setup:

1. Sequence of tokens $\{w_1, ..., w_i, ..., w_{n_L}\}$
2. Sequence of (trainable) embedding vectors $\{x_1, ..., x_i, ..., x_{n_L}\}$
3. Sequence of attention-transformed vectors $\{h_1, ..., h_i, ..., h_{n_L}\}$ with

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

Basic self-attention uses scaled dot product attention:

$$a(x_i, x_j) = \mathrm{softmax}\left(\frac{x_i \cdot x_j}{\sqrt{n_E}}\right) = \frac{\exp\left(\frac{x_i \cdot x_j}{\sqrt{n_E}}\right)}{\sum_{k=1}^{n_L} \exp\left(\frac{x_i \cdot x_k}{\sqrt{n_E}}\right)}$$

- the scaled dot-product $\frac{x_i \cdot x_j}{\sqrt{n_E}}$, normalized with softmax such that $\sum_j a(\cdot) = 1$.

- The self-attention transformation

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

with

$$a(x_i, x_j) = \text{softmax}(\frac{x_i \cdot x_j}{\sqrt{n_E}})$$

is a powerful architectural feature of transformers.

- The self-attention transformation

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

  with

$$a(x_i, x_j) = \mathrm{softmax}\left(\frac{x_i \cdot x_j}{\sqrt{n_E}}\right)$$

  is a powerful architectural feature of transformers.

Note:

- **basic self-attention has no learnable parameters**.
  - self-attention works indirectly through the word embeddings (more next slide)
- **basic self-attention ignores word order.**

- The self-attention transformation

$$h_i = \sum_{j=1}^{n_L} a(x_i, x_j) x_j$$

  with

$$a(x_i, x_j) = \text{softmax}(\frac{x_i \cdot x_j}{\sqrt{n_E}})$$

  is a powerful architectural feature of transformers.

Note:

- **basic self-attention has no learnable parameters**.
  - self-attention works indirectly through the word embeddings (more next slide)
- **basic self-attention ignores word order.**

The big initial gain from transformers, relative to RNNs, came from basic self-attention.

- But the successful models (e.g. BERT, GPT) do add parameters and word order information to $a(\cdot)$.

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

$$\boldsymbol{h}_i = a(x_i \cdot \boldsymbol{x}_{\text{the}})\boldsymbol{x}_{\text{the}} + a(x_i \cdot \boldsymbol{x}_{\text{cat}})\boldsymbol{x}_{\text{cat}} + \ldots + a(x_i \cdot \boldsymbol{x}_{\text{street}})\boldsymbol{x}_{\text{street}}$$

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

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

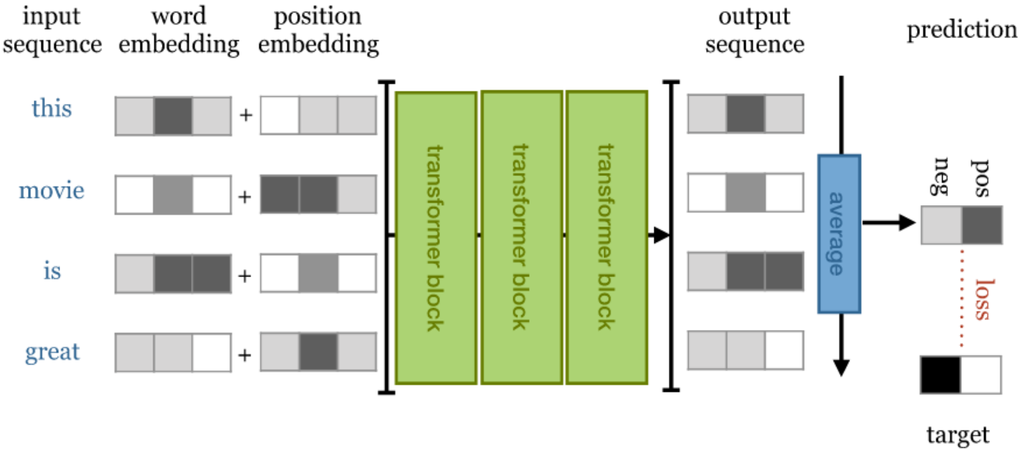
  where

$$\boldsymbol{h}_i = a(x_i \cdot \boldsymbol{x}_{\text{the}})\boldsymbol{x}_{\text{the}} + a(x_i \cdot \boldsymbol{x}_{\text{cat}})\boldsymbol{x}_{\text{cat}} + \ldots + a(x_i \cdot \boldsymbol{x}_{\text{street}})\boldsymbol{x}_{\text{street}}$$

Embedding layer will learn vectors $\boldsymbol{x}$ that tend to have **attention dot products** that contribute to the task at hand.

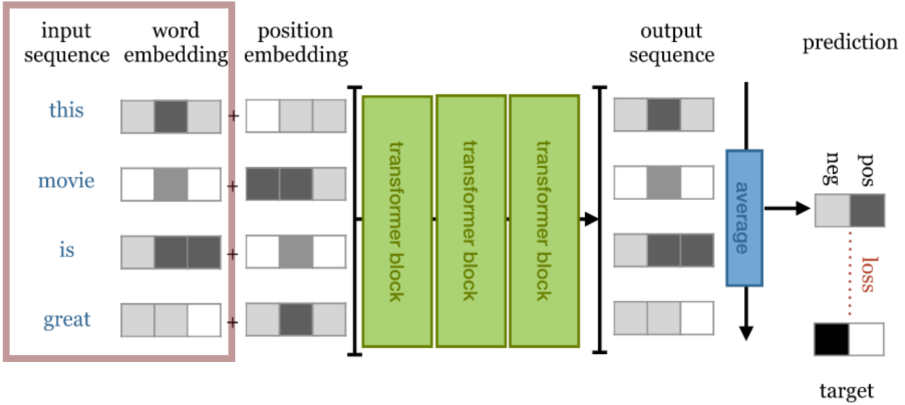
- For example, for most tasks, stopwords like "the" will not be helpful.
  - the learned embedding $\boldsymbol{x}_{\text{the}}$ will tend to have a low or negative dot product with more informative words.

Transformer Architecture: Sentiment Classification



Transformer for Sentiment Classification

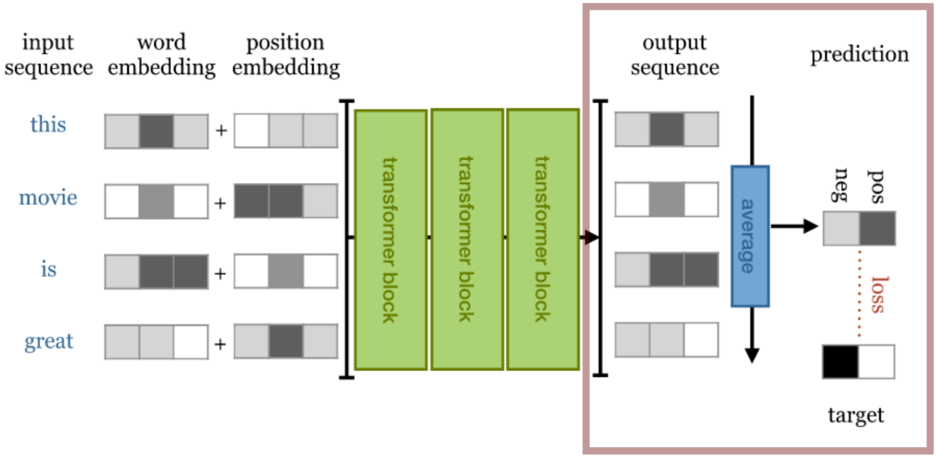
Input sequence → word embedding



- Input sequence of tokens $\{w_1, \ldots, w_i, \ldots, w_{n_L}\}$
- Trainable embedding vectors $[x_1 \ldots, x_i \ldots x_{n_L}]$

Transformer for Sentiment Classification

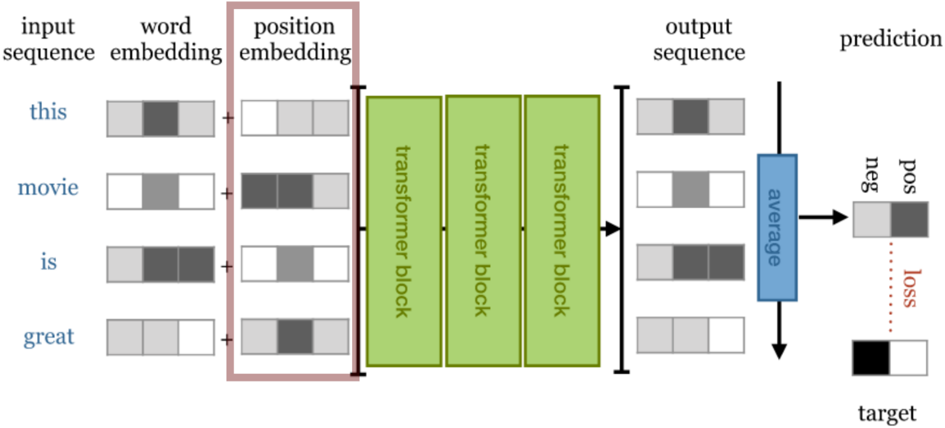
... → document embedding → sentiment score



- output sequence $\{h_1^y, \ldots, h_i^y, \ldots, h_{n_L}^y\}$
- averaged to produce **document vector** $\vec{d}$
- final output layer with sigmoid activation to produce probabilities $\hat{y}$ across positive and negative output classes.

Transformer for Sentiment Classification

... → position embedding → ...



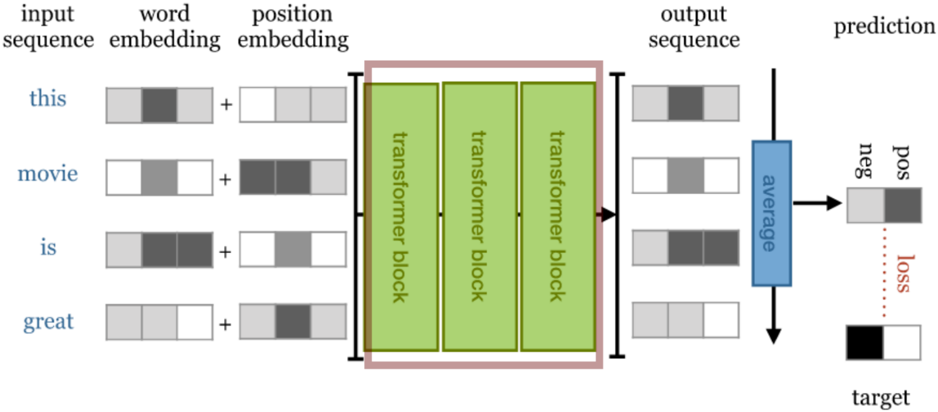
Position Embeddings

- To add word order information, transformers add a ***position embedding*** along with the ***word embedding*** as input to the attention layer.
- we have
  - word embeddings $\{x_1, \ldots, x_i, \ldots, x_{n_L}\}$ each with dimension n_E
  - position embeddings $\{t_1, \ldots, t_i, \ldots, t_{n_L}\}$, categorical embeddings for each position index i, also with dimension n_E.
- input to the first attention layer is element-wise addition of these embeddings,

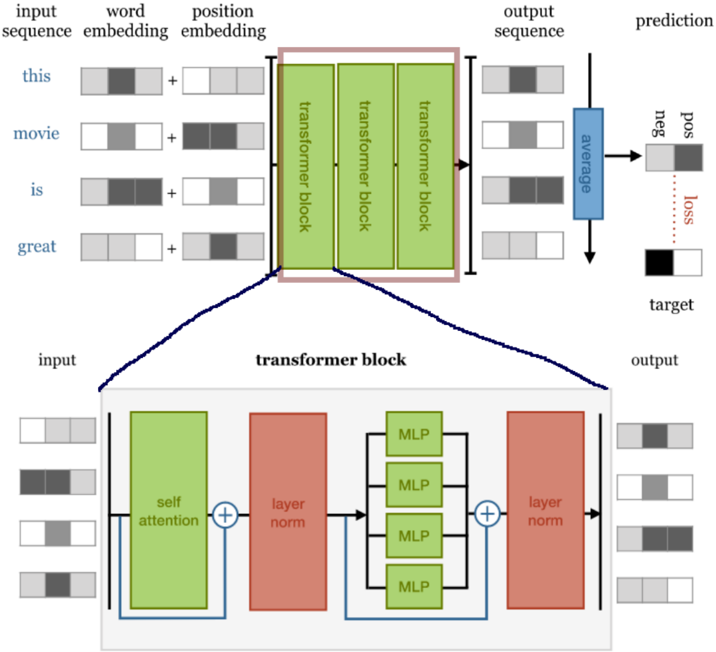
$$h^0_{1:n_L=}\{x_1 + t_1, \ldots, x_{n_L} + t_{n_L}\}$$

Transformer for Sentiment Classification

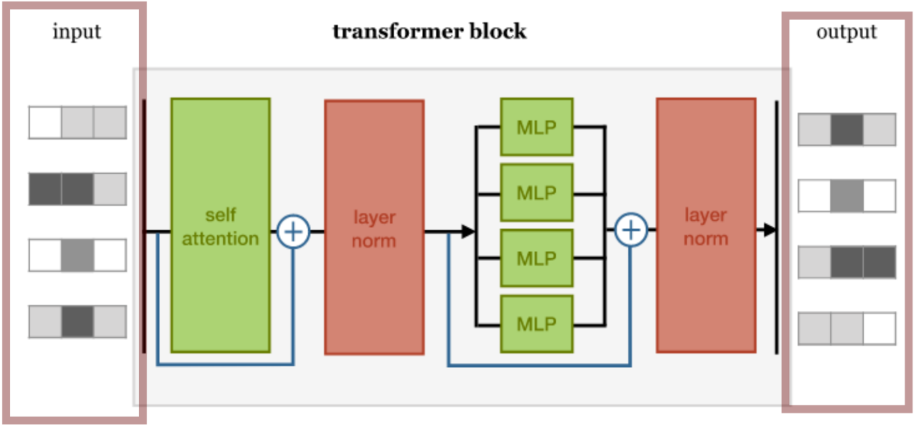
… → transformer blocks → …



A **transformer** consists of **stacked transformer blocks**

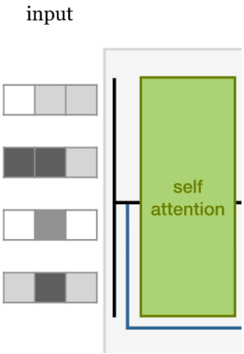


Transformer block (input and output)



- Each transformer block $l \in \{1, \ldots, n_y\}$ takes as input a sequence of vectors $h^{l-1}_{1:n_L}$ and outputs a sequence of vectors $h^{l}_{1:n_L}$, which become the input for the next transformer block.

Transformer Block (Self-Attention Layer)



the "self attention" layer:

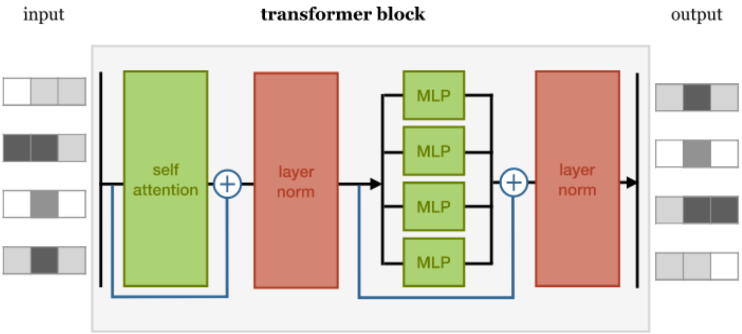
- input:
  - for the first block, includes the word embeddings summed with the position embeddings

$$h^0_{1:n_L} = \{x_1 + t_1, \ldots, x_{n_L} + t_{n_L}\}$$

  - for the later blocks, includes the output of the previous block h^{l-1}
- output:
  - matrix of self-attention-transformed vectors where item i is

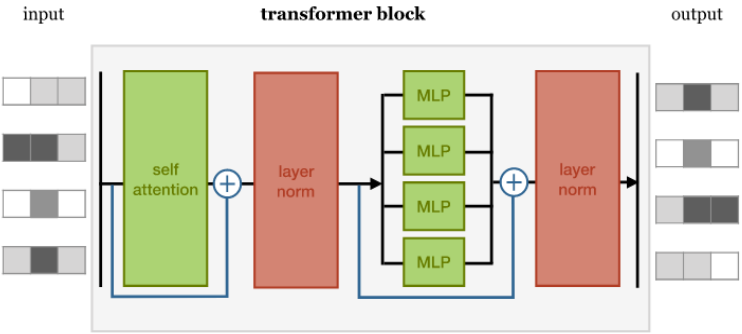
$$\sum_{j=1}^{n_L} a(h_i^{l-1}, h_j^{l-1}) h_j^{l-1}$$

Transformer Block (Residualization and Normalization)



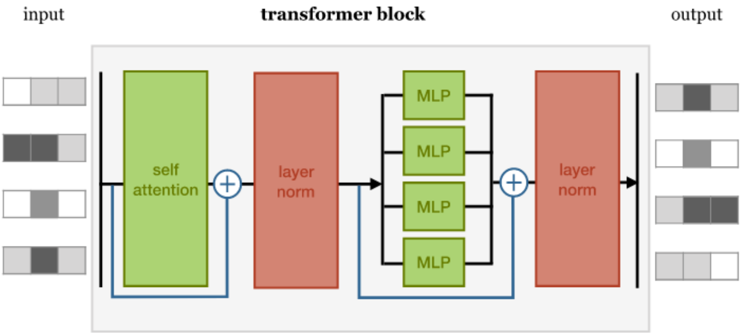
- self-attention layer's outputs are normalized
  - residual connections (blue line with $\oplus$) means that the input h^{l-1} is added element-wise to the output of the attention layer
    - model can "bypass" layer if its not adding value.
    - helps deep models learn faster.

Transformer Block (Residualization and Normalization)



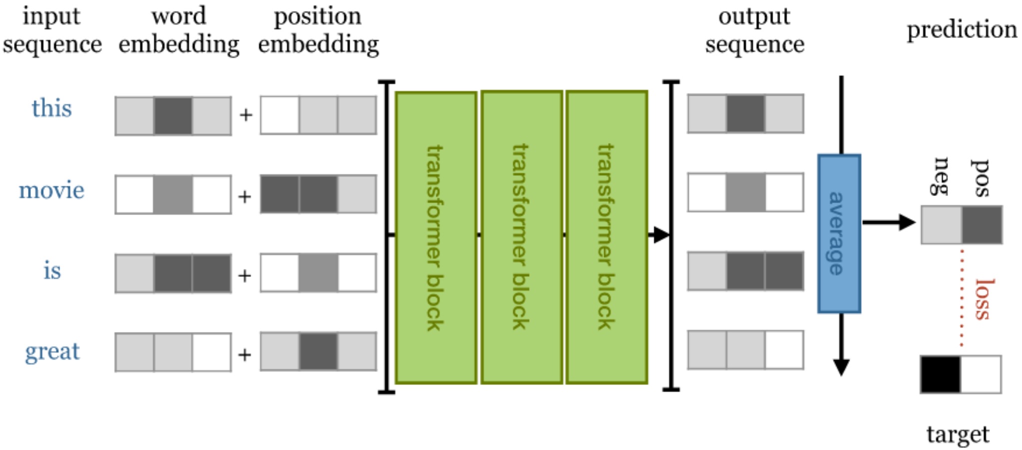
- self-attention layer's outputs are normalized
  - residual connections (blue line with $\oplus$) means that the input h^{l-1} is added element-wise to the output of the attention layer
    - model can "bypass" layer if its not adding value.
    - helps deep models learn faster.
  - "layer normalization": normalize the input vector for each data point to unit variance across dimensions.
    - distinct from *batch normalization*, which normalizes a feature to unit variance across a batch sample of data points.

Transformer Block (Dense MLP Layers)



- normalized self-attention outputs are piped to a multi-layer perceptron (MLP) with two hidden layers, with ReLU activation after the first layer.
- normalized again then output to h^{l+1}:
  - either to the next transformer block, or to the output layer h^{n_y}.

Transformer for Sentiment Classification



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

An interesting application

https://babel.poltextlab.com/

Video presentation: Prytkova et al, The employment impact of emerging digital technologies

Check for Understanding: True/False

1. A limitation of the Arora et al (2017) "tough-to-beat" sentence embeddings is that the vectors do not contain any information about word order.
2. Doc2Vec addresses the limits of the Arora et al (2017) embeddings by adding information on word order.
3. Unlike the other document embeddings, FastText embeddings (averaged hashed n-gram embeddings) do not have a geometric interpretation.