Text Data in Business and Economics

Basel University – Autumn 2023

9. Embedding Sequences with Attention

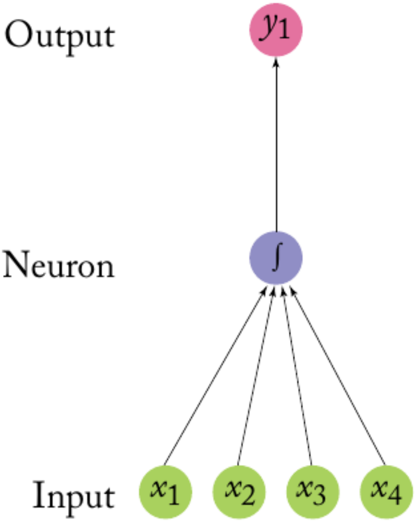
Outline

- Neural networks $\leftrightarrow$ deep learning models
  - solve machine learning problems, just like logistic regression or gradient boosted machines
  - use tensorflow, torch, or huggingface, rather than sklearn or xgboost.

- Neural networks $\leftrightarrow$ deep learning models
  - solve machine learning problems, just like logistic regression or gradient boosted machines
  - use tensorflow, torch, or huggingface, rather than sklearn or xgboost.
- **why use neural nets?**
  - sometimes outperform standard ML techniques on standard problems
  - greatly outperform standard ML techniques on some problems, for example language modeling

- Neural networks $\leftrightarrow$ deep learning models
  - solve machine learning problems, just like logistic regression or gradient boosted machines
  - use tensorflow, torch, or huggingface, rather than sklearn or xgboost.
- **why use neural nets?**
  - sometimes outperform standard ML techniques on standard problems
  - greatly outperform standard ML techniques on some problems, for example language modeling
- **why not use neural nets?**
  - usually worse than standard ML on standard problems
  - models are often more challenging/labor-intensive to implement
  - outputs are a black box and difficult to interpret
  - computational constraints: training requires specialized hardware.

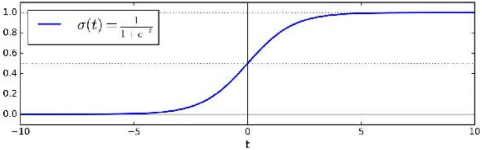
A "Neuron"



- applies dot product to vector of numerical inputs:
  - multiplies each input by a learned weight (parameter or coefficient)
  - sums these products
- applies a non-linear "activation function" to the sum
  - (e.g., the $\int$ shape indicates a sigmoid transformation)
- passes the output.

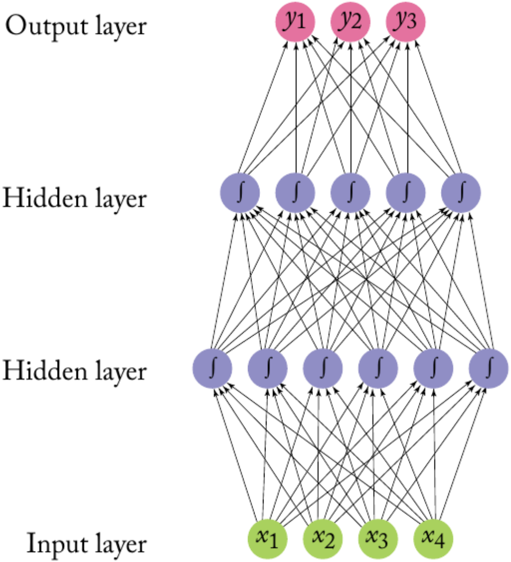
Logistic Regression ≈ "Neuron"

$$\hat{y} = \text{sigmoid}(\boldsymbol{x} \cdot \theta) = \frac{1}{1 + \exp(-\boldsymbol{x} \cdot \theta)}$$



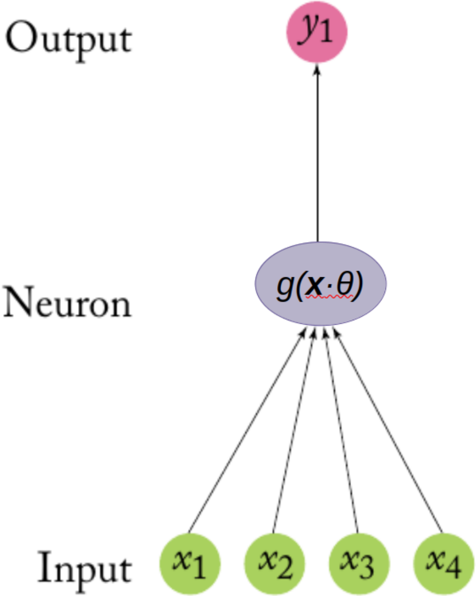
- applies dot product to vector of numerical inputs:
    - multiplies each input by a learned weight (parameter or coefficient)
    - sums these products
- applies a non-linear "activation function" (sigmoid) to the sum
- passes the output.

Multi-Layer Perceptron (MLP)



- A multilayer perceptron (also called a feed-forward network or sequential model) stacks neurons horizontally and vertically.
- alternatively, think of it as a stacked ensemble of logistic regression models.
- this vertical stacking is the "deep" in "deep learning"!

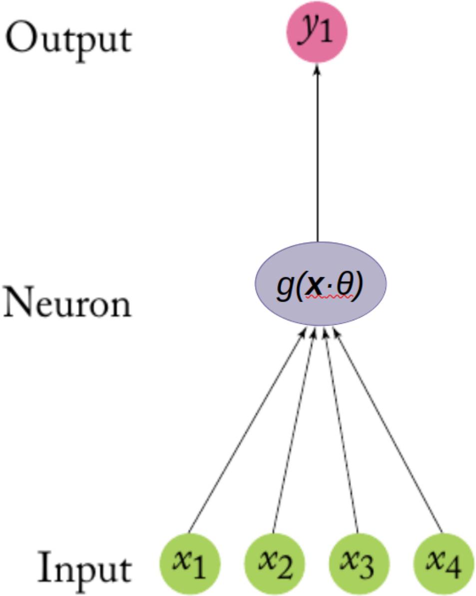
Activation functions $g(\boldsymbol{x} \cdot \theta)$



Previously we had

$$g(\boldsymbol{x} \cdot \theta) = \text{sigmoid}(\boldsymbol{x} \cdot \theta) = \frac{1}{1 + \exp(-\boldsymbol{x} \cdot \theta)}$$

Activation functions $g(\boldsymbol{x} \cdot \theta)$

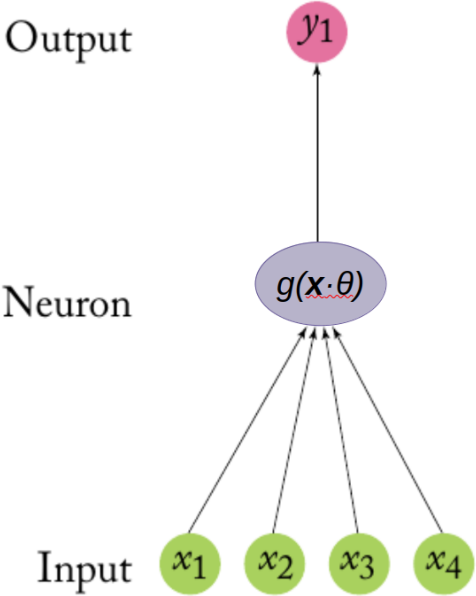


Previously we had

$$g(\boldsymbol{x} \cdot \theta) = \text{sigmoid}(\boldsymbol{x} \cdot \theta) = \frac{1}{1 + \exp(-\boldsymbol{x} \cdot \theta)}$$

But – it turns out that sigmoid does not work well in hidden layers, mainly because gradient is flat except around zero.

Activation functions $g(\boldsymbol{x} \cdot \theta)$



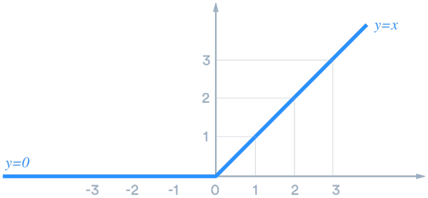
Previously we had

$$g(\boldsymbol{x} \cdot \theta) = \text{sigmoid}(\boldsymbol{x} \cdot \theta) = \frac{1}{1 + \exp(-\boldsymbol{x} \cdot \theta)}$$

But – it turns out that sigmoid does not work well in hidden layers, mainly because gradient is flat except around zero.

ReLU (rectified linear unit) function:

$$g(\boldsymbol{x} \cdot \theta) = \text{ReLU}(\boldsymbol{x} \cdot \theta) = \max\{0, \boldsymbol{x} \cdot \theta\}$$



Equation Notation: Multi-Layer Perceptron

- An multi-layer perceptron (MLP) with two hidden layers is

$$\boldsymbol{y} = \boldsymbol{g}_2(\boldsymbol{g}_1(\boldsymbol{x} \cdot \boldsymbol{\omega}_1) \cdot \boldsymbol{\omega}_2) \cdot \boldsymbol{\omega}_y$$

$$\boldsymbol{y} \in \{0,1\}^{n_y}, \boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega}_1 \in \mathbb{R}^{n_x \times n_1}, \boldsymbol{\omega}_2 \in \mathbb{R}^{n_1 \times n_2}, \boldsymbol{\omega}_y \in \mathbb{R}^{n_2 \times n_y}$$

  - $n_1, n_2 =$ dimensionality in first and second hidden layer.
  - $\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \boldsymbol{\omega}_y =$ set of learnable weights for the first hidden, second hidden, and output layer
  - $\boldsymbol{g}_1(\cdot), \boldsymbol{g}_2(\cdot) =$ element-wise non-linear functions (typically ReLU) for first and second layer.

Equation Notation: Multi-Layer Perceptron

- An multi-layer perceptron (MLP) with two hidden layers is

$$\boldsymbol{y} = \boldsymbol{g}_2(\boldsymbol{g}_1(\boldsymbol{x} \cdot \boldsymbol{\omega}_1) \cdot \boldsymbol{\omega}_2) \cdot \boldsymbol{\omega}_y$$

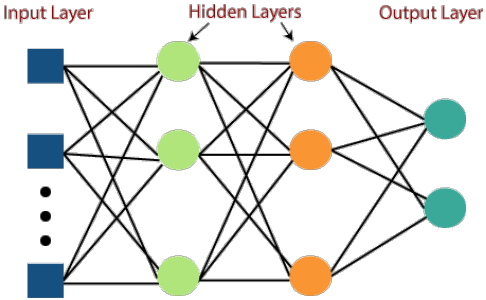
$$\boldsymbol{y} \in \{0,1\}^{n_y}, \boldsymbol{x} \in \mathbb{R}^{n_x}, \boldsymbol{\omega}_1 \in \mathbb{R}^{n_x \times n_1}, \boldsymbol{\omega}_2 \in \mathbb{R}^{n_1 \times n_2}, \boldsymbol{\omega}_y \in \mathbb{R}^{n_2 \times n_y}$$

  - n_1, n_2 = dimensionality in first and second hidden layer.
  - $\boldsymbol{\omega}_1, \boldsymbol{\omega}_2, \boldsymbol{\omega}_y$ = set of learnable weights for the first hidden, second hidden, and output layer
  - $\boldsymbol{g}_1(\cdot), \boldsymbol{g}_2(\cdot)$ = element-wise non-linear functions (typically ReLU) for first and second layer.

- Can also be written in decomposed notation:

$$\boldsymbol{h}_1 = \boldsymbol{g}_1(\boldsymbol{x} \cdot \boldsymbol{\omega}_1)$$
$$\boldsymbol{h}_2 = \boldsymbol{g}_2(\boldsymbol{h}_1 \cdot \boldsymbol{\omega}_2)$$
$$\boldsymbol{y} = \boldsymbol{h}_2 \cdot \boldsymbol{\omega}_y$$

where $\boldsymbol{h}_l$ indicate hidden layers.



Outline

What is an Embedding?

- In a broad sense, **“embedding”** refers to a lower-dimensional dense vector representation of a higher-dimensional object.
  - in NLP, this higher-dimensional object could be a word, phrase, sentence, or document.

What is an Embedding?

- In a broad sense, **“embedding”** refers to a lower-dimensional dense vector representation of a higher-dimensional object.
  - in NLP, this higher-dimensional object could be a word, phrase, sentence, or document.
- Not embeddings:
  - counts over LIWC dictionary categories
  - sklearn CountVectorizer count vectors

What is an Embedding?

- In a broad sense, **“embedding”** refers to a lower-dimensional dense vector representation of a higher-dimensional object.
  - in NLP, this higher-dimensional object could be a word, phrase, sentence, or document.
- Not embeddings:
  - counts over LIWC dictionary categories
  - sklearn CountVectorizer count vectors
- Embeddings:
  - PCA reductions of the word count vectors
  - LDA topic shares
  - word embeddings from GloVe

Categorical Embeddings = dense representations of categorical variables

Say we have a binary classification problem with outcome Y:

- we have a high-dimensonal categorical variable (e.g. area of law with 1000 categories)
- including dummy variables A for each category in your ML model is computationally expensive.

Categorical Embeddings = dense representations of categorical variables

Say we have a binary classification problem with outcome Y:

- we have a high-dimensonal categorical variable (e.g. area of law with 1000 categories)
- including dummy variables A for each category in your ML model is computationally expensive.

Embedding approaches:

1. PCA applied to the dummy variables A to get lower-dimensional $\tilde{A}$.

Categorical Embeddings = dense representations of categorical variables

Say we have a binary classification problem with outcome Y:

- we have a high-dimensonal categorical variable (e.g. area of law with 1000 categories)
- including dummy variables A for each category in your ML model is computationally expensive.

Embedding approaches:

1. PCA applied to the dummy variables A to get lower-dimensional $\tilde{A}$.
2. Regress Y on A, predict $\hat{Y}(A_i)$, add that as a predictor in your model instead.

Categorical Embeddings = dense representations of categorical variables

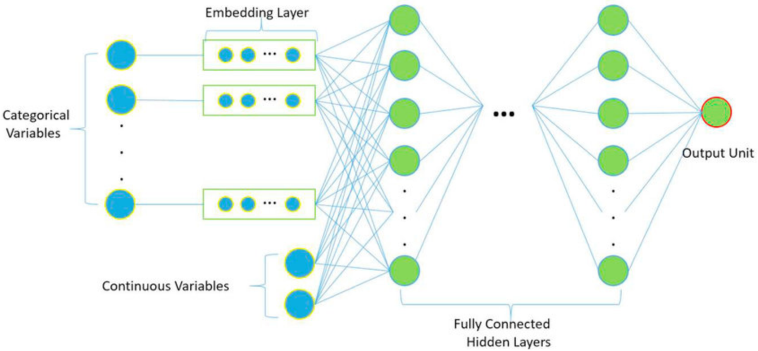
Say we have a binary classification problem with outcome Y:

- we have a high-dimensonal categorical variable (e.g. area of law with 1000 categories)
- including dummy variables A for each category in your ML model is computationally expensive.

Embedding approaches:

1. PCA applied to the dummy variables A to get lower-dimensional $\tilde{A}$.
2. Regress Y on A, predict $\hat{Y}(A_i)$, add that as a predictor in your model instead.

(2) is quite close to what embedding layers do in neural nets.



An embedding layer is efficient matrix multiplication:

$$\underbrace{h_1}_{n_E \times 1} = \underbrace{\omega_E}_{n_E \times n_w} \cdot \underbrace{x}_{n_x \times 1}$$

- x = a categorical variable (e.g., representing a word)
  - one-hot vector with a single item equaling one. Input to the embedding layer.
- h_1 = the first hidden layer of the neural net
  - The output of the embedding layer.

The embedding matrix ω_E encodes predictive information about the categories; it has a spatial interpretation when projected to two dimensions.

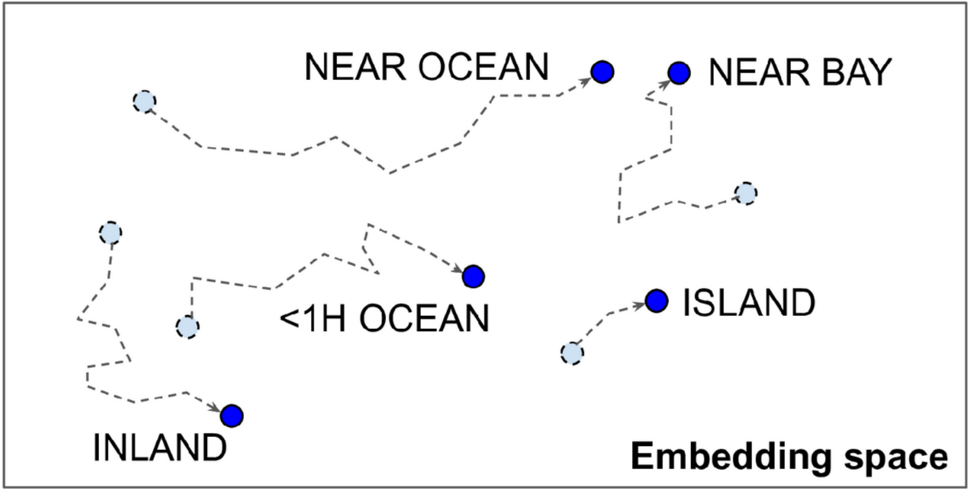


Figure 13-4. Embeddings will gradually improve during training

Word Embeddings = NN layers mapping word indexes to dense vectors

Word Embeddings = NN layers mapping word indexes to dense vectors

- Each document i is a list of word indexes $\{w_{i1}, ..., w_{it}, ..., w_{in_i}\}$.
  - Let W_i be the matrix of one-hot vectors (dummy variables) for each token position in the document
  - W_i is extremely high-dimensional – not usable for machine learning

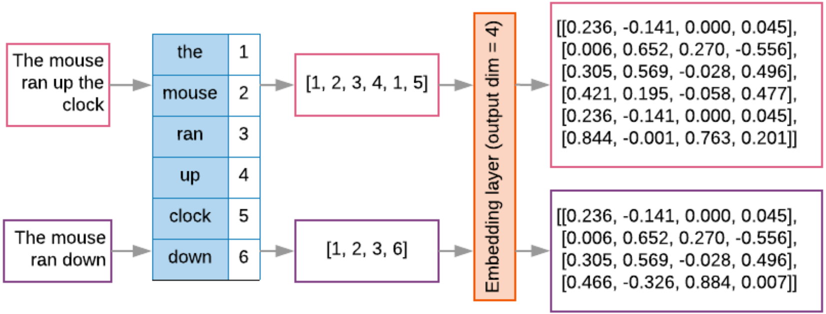
Word Embeddings = NN layers mapping word indexes to dense vectors

- Each document i is a list of word indexes $\{w_{i1}, ..., w_{it}, ..., w_{in_i}\}$.
  - Let W_i be the matrix of one-hot vectors (dummy variables) for each token position in the document
  - W_i is extremely high-dimensional – not usable for machine learning
- Rather than use W_i, define $X_i = EW_i$, where E is a learnable embedding matrix.
  - each item $x_{i1}, ..., x_{it}$ is a low-dimensional dense embedding representation for each word.

Word Embeddings = NN layers mapping word indexes to dense vectors

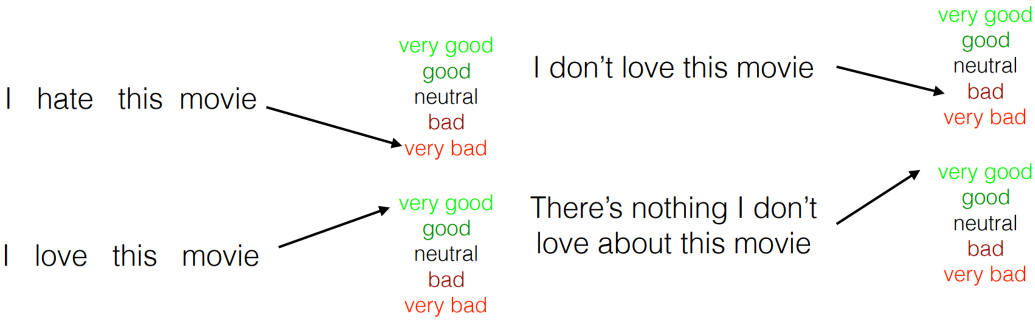
- Each document i is a list of word indexes $\{w_{i1}, ..., w_{it}, ..., w_{in_i}\}$.
  - Let W_i be the matrix of one-hot vectors (dummy variables) for each token position in the document
  - W_i is extremely high-dimensional – not usable for machine learning

- Rather than use W_i, define $X_i = EW_i$, where E is a learnable embedding matrix.
  - each item $x_{i1}, ..., x_{it}$ is a low-dimensional dense embedding representation for each word.



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.

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** $\boldsymbol{x}$, 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 $\boldsymbol{x}$**, take as input a **sequence of tokens** $\{w_1, ..., w_t, ...w_n\}$.
- "Traditional" architectures:
  - Convolutional neural nets (CNNs)
  - Recurrent Neural Nets (RNNs)
- Since 2018, CNNs and RNNs (as currently implemented) usually get worse performance than transformers (attentional neural nets).

Universal Sentence Encoder (USE) Produces Embeddings that are Sensitive to Word Order and Context

```python
import tensorflow_hub as hub

embed = hub.Module("https://tfhub.dev/google/"
    "universal-sentence-encoder/1")

embedding = embed([
    "The quick brown fox jumps over the lazy dog."])
```

Listing 1: Python example code for using the universal sentence encoder.

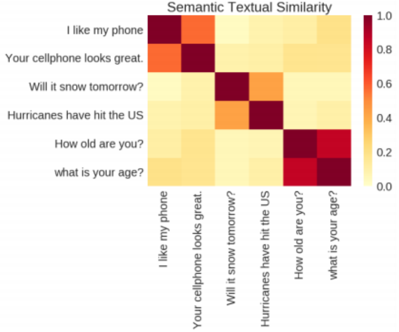


Figure 1: Sentence similarity scores using embeddings from the universal sentence encoder.

- Neural net architecture with embeddings pre-trained on:
  - Identifying co-occuring sentences
  - Identifying message-response pairs (Henderson et al 2017)

Multilingual Encoders

- The multilingual sentence encoder (**MUSE**) expands the USE model to sixteen languages, in a single embedding model.
  - Trained on a similar array of tasks in all languages, so that it can be used out-of-the-box.

Multilingual Encoders

- The multilingual sentence encoder (**MUSE**) expands the USE model to sixteen languages, in a single embedding model.
  - Trained on a similar array of tasks in all languages, so that it can be used out-of-the-box.
- Facebook's LASER encoder produces vectors for 90 languages with a single model.
  - bidirectional LSTM architecture
  - trained on multilingual machine translation task

Sentence-BERT

- The document embeddings produced by BERT do not perform well for sentence similarity tasks.
- S-BERT (Reimers and Gurevych 2019):
  - fine-tune BERT embeddings to classify sentence pairs in textual entailment task.
  - significantly improves performance of sentence embeddings on standard tasks.

SentenceTransformers

- SentenceTransformers (sbert.net) is an amazing python package for embedding texts or short documents.
- Initially based on S-BERT but expanded to many additional models, including embeddings trained on other tasks besides entailment:
  - paraphrase identification
  - semantic textual similarity
  - duplicate question detection
  - question-answer retrieval
- monolingual and multilingual models (for over 100 languages)

Transformers

- Since a 2017 paper (Vaswani et al 2017), deep learning for NLP has been transformed by a new class of models: **transformers.**

Transformers

- Since a 2017 paper (Vaswani et al 2017), deep learning for NLP has been transformed by a new class of models: **transformers.**
- Standard approach:
  - represent documents as counts over words/phrases, shares over topics, or the average of word embeddings.

Transformers

- Since a 2017 paper (Vaswani et al 2017), deep learning for NLP has been transformed by a new class of models: **transformers.**
- Standard approach:
  - represent documents as counts over words/phrases, shares over topics, or the average of word embeddings.
- Recurrent neural nets can process whole documents word-by-word:
  - but they have to sweep through the whole document at each training epoch, so they learn too slowly.

Transformers

- Since a 2017 paper (Vaswani et al 2017), deep learning for NLP has been transformed by a new class of models: **transformers.**
- Standard approach:
  - represent documents as counts over words/phrases, shares over topics, or the average of word embeddings.
- Recurrent neural nets can process whole documents word-by-word:
  - but they have to sweep through the whole document at each training epoch, so they learn too slowly.
- Transformers overcome these limitations:
  - intuitively, they provide a way to efficiently read in an entire document and learn the meaning of all words and all interactions between words.

Self-Attention – the fundamental computation underlying transformers

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

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

- In previous models, the sequence $x_{1:n_L}$ could be flattened to an $n_L n_E$-dimensional vector and piped to the hidden layers for use in the task, e.g. sentiment classification.

Self-Attention – the fundamental computation underlying transformers

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

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

- In previous models, the sequence $x_{1:n_L}$ could be flattened to an $n_L n_E$-dimensional vector and piped to the hidden layers for use in the task, e.g. sentiment classification.
- A **self-attention layer** transforms $x_{1:n_L}$ into a second sequence $h_{1:n_L}$, where

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

  - 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.
- $h_{1:n_L}$ is flattened and piped to the network's hidden layers, rather than $x_{1:n_L}$.

Basic Self-Attention

Setup:

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

$$a(x_i, x_j) = \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)}$$

- the dot-product $x_i \cdot x_j$, normalized with softmax such that $\sum_j a(\cdot) = 1$.

Basic Self-Attention

Setup:

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

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

Basic self-attention specifies

$$a(x_i, x_j) = \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)}$$

- the dot-product $x_i \cdot x_j$, normalized with softmax such that $\sum_j a(\cdot) = 1$.
- Putting it together:

$$h_i = \sum_{j=1}^{n_L} \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)} x_j$$

- The basic self-attention transformation

$$h_i = \sum_{j=1}^{n_L} \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)} x_j$$

is the foundational ingredient of transformers.

- The basic self-attention transformation

$$h_i = \sum_{j=1}^{n_L} \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)} x_j$$

  is the foundational ingredient of transformers.

Note the following simplifications:

- **basic self-attention has no learnable parameters**.
  - self-attention works indirectly through allowing the word embeddings to interact with each other
- **basic self-attention ignores word order.**

- The basic self-attention transformation

$$h_i = \sum_{j=1}^{n_L} \frac{\exp(x_i \cdot x_j)}{\sum_{k=1}^{n_L} \exp(x_i \cdot x_k)} x_j$$

  is the foundational ingredient of transformers.

Note the following simplifications:

- **basic self-attention has no learnable parameters**.
  - self-attention works indirectly through allowing the word embeddings to interact with each other
- **basic self-attention ignores word order.**

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

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

Self-attention allows words to interact with each other

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

Self-attention allows words to interact with each other

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

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

- For example, most transformers are pre-trained on a language modeling task (predicting a left-out word or sentence)
- in this task, 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.

Autoregressive vs Autoencoding Language Models

- **Autoregressive models**:
  - e.g. **GPT = "Generative Pre-Trained Transformer"**:
  - pretrained on classic language modeling task: guess the next token having read all the previous ones.
  - during training, attention heads only view previous tokens, not subsequent tokens.
  - ideal for text generation.
- **Autoencoding models**
  - e.g. **BERT = "Bidirectional Encoder Representations from Transformers"**
  - pretrained by dropping/shuffling input tokens and trying to reconstruct the original sequence.
  - usually build bidirectional representations and get access to the full sequence.
  - can be fine-tuned and achieve great results on many tasks, e.g. text classification.

Shortcut: Using BERT-Based Pre-Trained Models

Shortcut: Using BERT-Based Pre-Trained Models

```
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.

BERT (and RoBERTa)

- BERT = Bidirectional Encoder Representations from Transformers
  - RoBERTa = Robust BERT
- Architecture:
  - a stack of transformer blocks with a self-attention layer and an MLP.
  - The largest BERT model has 24 blocks, embedding dimension of 1024, and 16 attention heads.
    $\approx$ 340M parameters to learn.

BERT (and RoBERTa)

- BERT = Bidirectional Encoder Representations from Transformers
  - RoBERTa = Robust BERT
- Architecture:
  - a stack of transformer blocks with a self-attention layer and an MLP.
  - The largest BERT model has 24 blocks, embedding dimension of 1024, and 16 attention heads.
    $\approx$ 340M parameters to learn.
- Task: Masked language modeling:
  - 15% of words masked
  - if masked: replace with [MASK] 80% of the time, a random token 10% of the time, and left unchanged 10% of the time.
  - model has to predict the original word.

BERT (and RoBERTa)

- BERT = Bidirectional Encoder Representations from Transformers
  - RoBERTa = Robust BERT
- Architecture:
  - a stack of transformer blocks with a self-attention layer and an MLP.
  - The largest BERT model has 24 blocks, embedding dimension of 1024, and 16 attention heads.
    $\approx$ 340M parameters to learn.
- Task: Masked language modeling:
  - 15% of words masked
  - if masked: replace with [MASK] 80% of the time, a random token 10% of the time, and left unchanged 10% of the time.
  - model has to predict the original word.
- Unlike GPT, BERT attention observes all tokens in the sequence, reads backwards and forwards (bidirectional).

BERT (and RoBERTa)

- BERT = Bidirectional Encoder Representations from Transformers
  - RoBERTa = Robust BERT
- Architecture:
  - a stack of transformer blocks with a self-attention layer and an MLP.
  - The largest BERT model has 24 blocks, embedding dimension of 1024, and 16 attention heads.
    $\approx$ 340M parameters to learn.
- Task: Masked language modeling:
  - 15% of words masked
  - if masked: replace with [MASK] 80% of the time, a random token 10% of the time, and left unchanged 10% of the time.
  - model has to predict the original word.
- Unlike GPT, BERT attention observes all tokens in the sequence, reads backwards and forwards (bidirectional).
- Corpus:
  - 800M words from English books (modern work, from unpublished authors), by Zhu et al (2015).
  - 2.5B words of text from English Wikipedia articles (without markup).

Application: Climate-Related Corporate Disclosures (Bingler, Kraus, and Leippold 2021)

- Fine-tunes RoBERTa ("Robust BERT") to classify texts related to corporate climate disclosures (using hand-annotated sample).

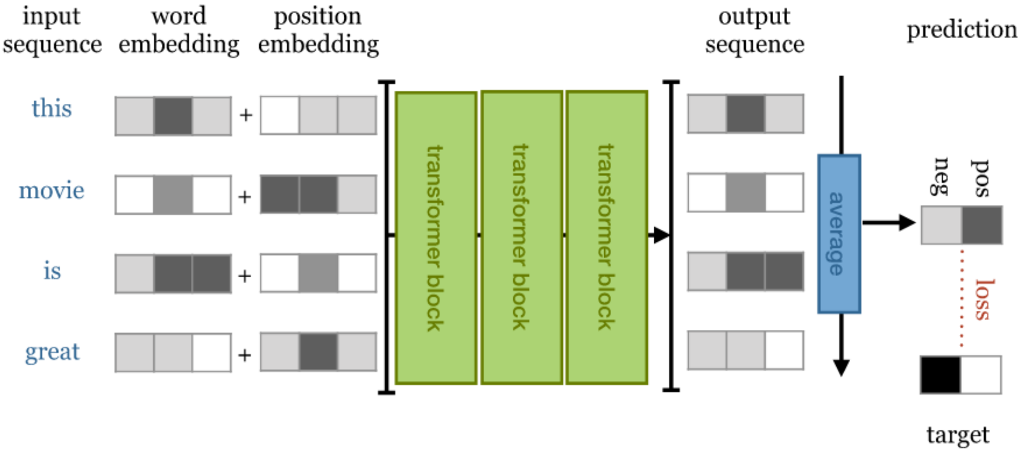
Table 3. Out-of-sample performance comparison between baseline models and our proposed ClimateBERT. Performance is reported in precision for each category.

	Governance	Strategy	Risk Management	Metrics & Targets	General Language	Overall Accuracy
Tf-idf	0.43	0.00	0.40	0.35	0.00	0.24
Sentence Enc.	0.19	0.57	0.15	0.24	0.00	0.23
RoBERTa Para.	0.26	0.25	0.25	0.25	0.07	0.22
RoBERTa Sent.	0.96	0.92	0.84	0.74	0.32	0.75
ClimateBERT	0.94	0.90	0.79	0.77	0.65	0.81

- model applied to large sample, shows that most disclosures are about more subjective / less verifiable aspects of climate disclosures.

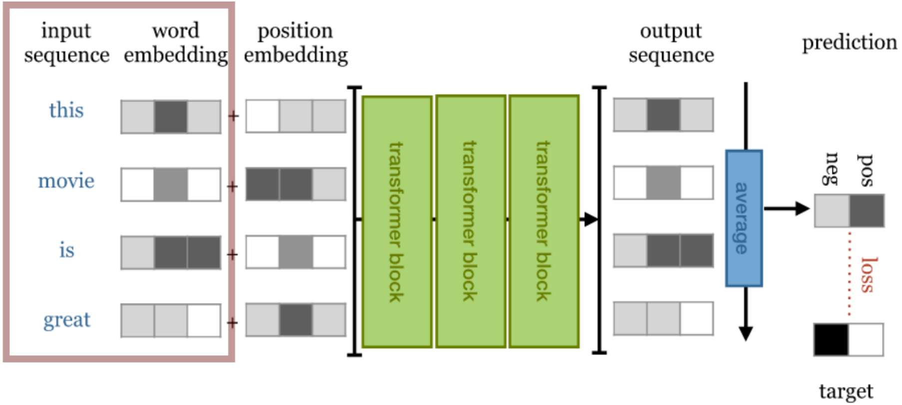
Outline

Transformer for Sentiment Classification



Transformer for Sentiment Classification

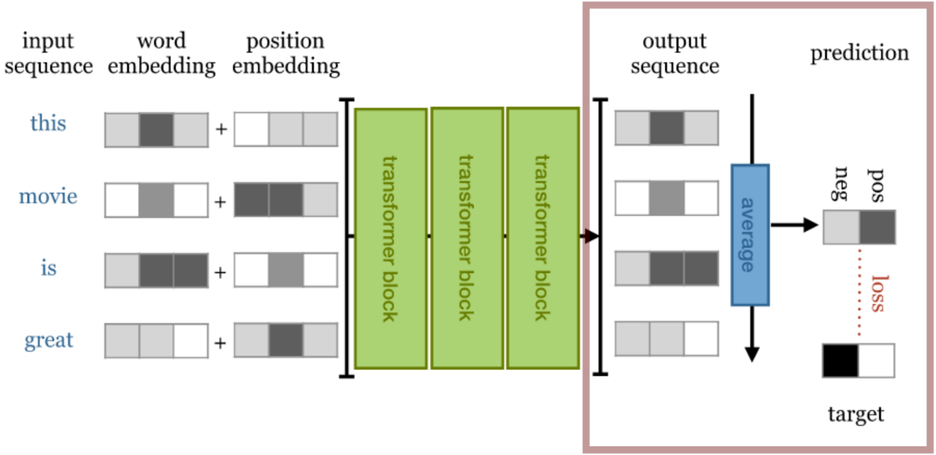
Input sequence → word embedding



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

Transformer for Sentiment Classification

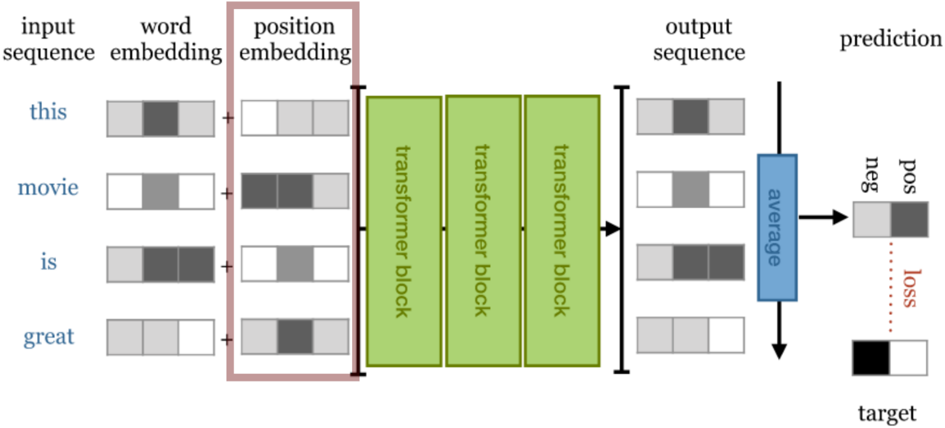
... → document embedding → sentiment score



- output sequence $\{h_1^y, \ldots, h_i^y, \ldots, h_{n_L}^y\}$
- averaged to produce **document vector** $\vec{d}$

Transformer for Sentiment Classification

... → position embedding → ...



Position Embeddings

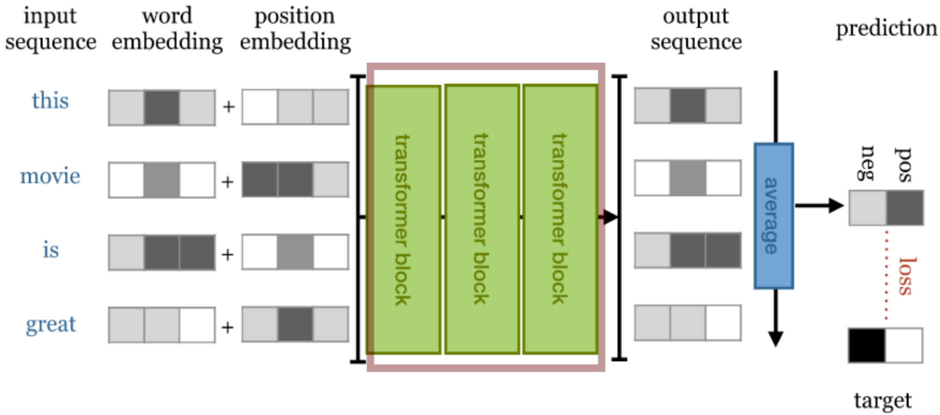
- To add word order information, transformers add a ***position embedding*** along with the ***word embedding*** as input to the attention layer.
- input to transformer block is

$$h^0 = \begin{bmatrix} x_1 & \dots & x_i & \dots & x_{n_L} \\ t_1 & \dots & t_i & \dots & t_{n_L} \end{bmatrix}$$

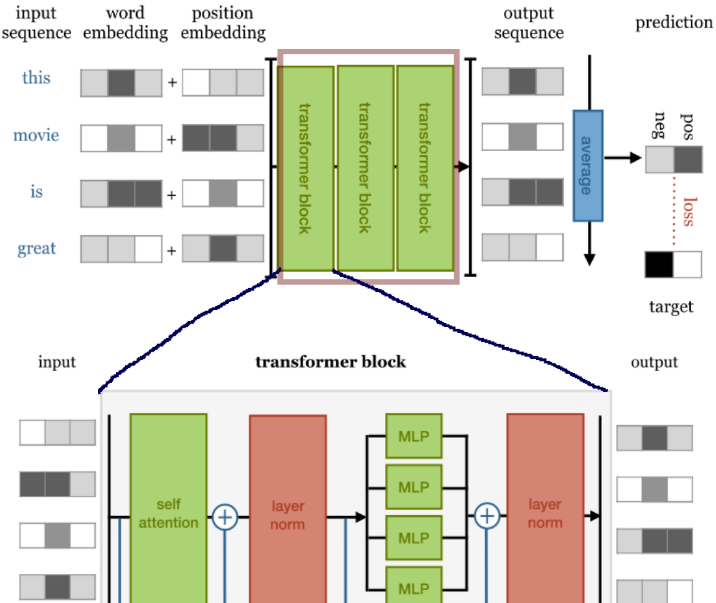
  which includes
  - word embeddings $\{x_1, ..., x_i, ..., x_{n_L}\}$ with dimension n_E
  - stacked with $\{t_1, ..., t_i, ..., t_{n_L}\}$, learnable categorical embeddings with dimension n_t for each index number i itself.
- Note:
  - puts a hard limit on sequence lengths
  - Positional encodings (or any direct information on word order) often not necessary after all (Irie et al 2019; Schlag et al 2021, Sinha et al 2021).

Transformer for Sentiment Classification

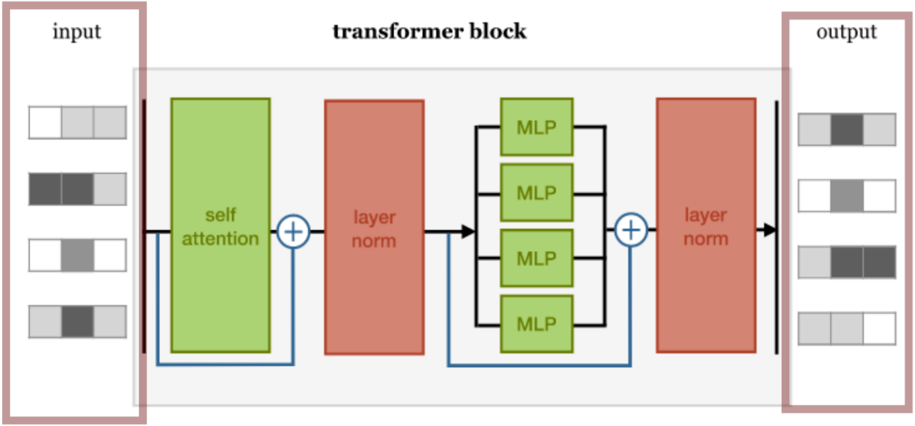
... → transformer blocks → ...



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

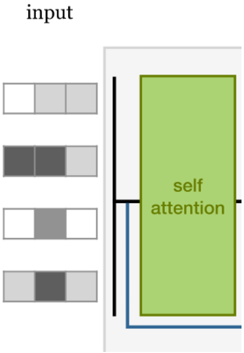


Transformer block (input and output)



- Each transformer block $l \in \{0, ..., n_y\}$ takes as input a sequence of vectors $h^l_{1:n_L}$ 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)

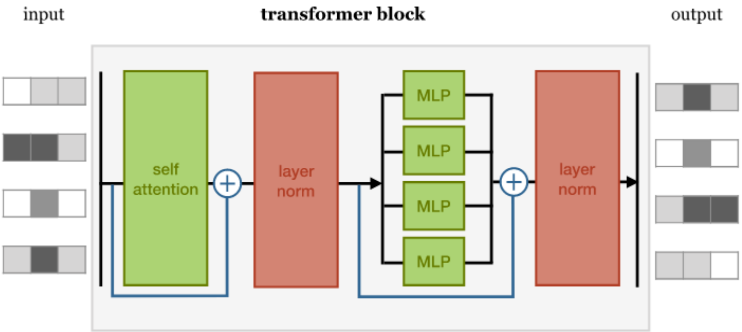


the "self attention" layer:

- input:
  - for the first block, includes the word embeddings and position embeddings h^0
  - for the later blocks, includes the output of the previous block h^l
- output:
  - matrix of self-attention-transformed vectors where item i is

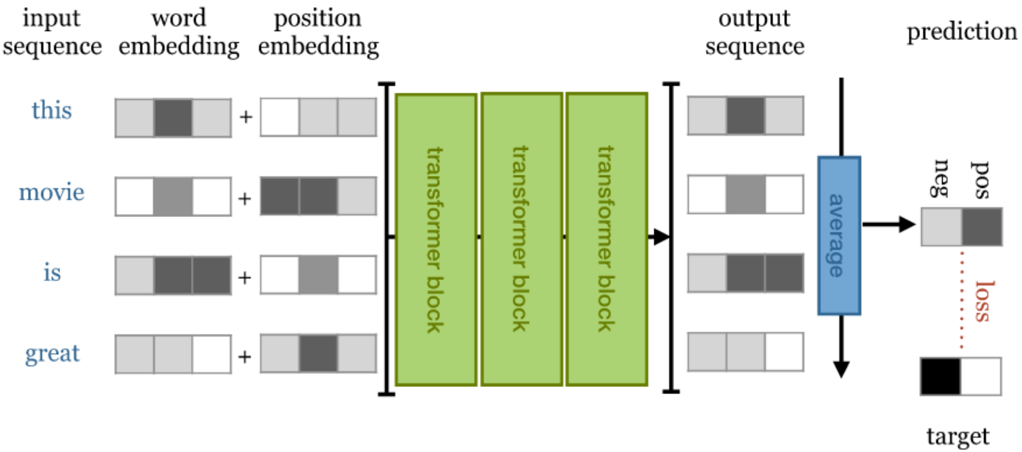
$$\sum_{j=1}^{n_L} a(h_i^l, h_j^l) h_j^l$$

The Transformer Block (Dense Layers)



- self-attention layer's outputs are normalized
  - we will come back to residual connections (blue line with $\oplus$) and "layer normalization" next week.
- 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.