Statistical Language Modeling
From N-grams to Transformers

Gustave Cortal
Table of contents

N-grams

Feedforward neural networks

Recurrent neural networks and attention mechanisms

Transformers
Tokenization

Tokenization is splitting text into individual words or **tokens**

Multiple challenges:

- Different delimiters: spaces, punctuation
- Contractions: "can’t" → "can not"
- Special cases: dates, numbers, URLs, hashtags, email addresses
Tokenization

**Input**

"Natural language processing enables computers to understand human language."

**Tokenized output**

Natural, language, processing, enables, computers, to, understand, human, language, .
Input
"Dr. Smith’s email, dr.smith@example.com, isn’t working since 01/02/2023; try reaching out at (555) 123-4567 in San Francisco."

Tokenized output
Dr., Smith's, email, ,, dr.smith@example.com, ,, isn't, working, since, 01/02/2023, ,, try, reaching, out, at, (, 555, ), 123-4567, in, San Francisco, .
Tokenization

**Rule-based approach**
Use predefined rules, like splitting by spaces or punctuation using regular expressions

**Machine learning approach**
Learn from data to handle complex cases, e.g., using Byte-Pair Encoding subword tokenization
N-grams

Gustave Cortal
A **language model** is a probabilistic model that:

- computes the **probability of a sequence of words** \( S \)
  \[ P(S) = P(w_1, w_2, ..., w_n) \]
- computes the **probability of an upcoming word**
  \[ P(w_5 | w_1, w_2, w_3, w_4) \]

Useful for building conversational agents, performing translation, speech recognition, summarization, question-answering, classification, etc.

For example, for speech recognition:

\[ P(\text{I saw a van}) \rightarrow P(\text{eyes awe of an}) \]
What is a language model?

A language model is a probabilistic model that:

- computes the probability of a sequence of words $S$
  $$P(S) = P(w_1, w_2, \ldots, w_n)$$
- computes the probability of an upcoming word
  $$P(w_5|w_1, w_2, w_3, w_4)$$

Useful for building conversational agents, performing translation, speech recognition, summarization, question-answering, classification, etc.
What is a language model?

A language model is a probabilistic model that:

- computes the probability of a sequence of words $S$
  
  $$P(S) = P(w_1, w_2, ..., w_n)$$

- computes the probability of an upcoming word
  
  $$P(w_5 | w_1, w_2, w_3, w_4)$$

Useful for building conversational agents, performing translation, speech recognition, summarization, question-answering, classification, etc.

For example, for speech recognition:

$$P(I \text{ saw a van}) \gg P(\text{eyes awe of an})$$
How to compute $P(S)$?

Definition of **conditional probabilities**:

$$P(B|A) = \frac{P(A, B)}{P(A)}$$

$$P(A, B) = P(A)P(B|A)$$
How to compute $P(S)$?

Definition of **conditional probabilities**:

$$P(B|A) = \frac{P(A, B)}{P(A)}$$
$$P(A, B) = P(A)P(B|A)$$

Applying the **chain rule** to multiple variables:

$$P(A, B, C, D) = P(A)P(B|A)P(C|A, B)P(D|A, B, C)$$
How to compute $P(S)$?

Definition of **conditional probabilities**:

$$P(B|A) = P(A, B)/P(A)$$
$$P(A, B) = P(A)P(B|A)$$

Applying the **chain rule** to multiple variables:

$$P(A, B, C, D) = P(A)P(B|A)P(C|A, B)P(D|A, B, C)$$

Applying the chain rule to compute the joint probability of words in a sentence:

$$P(I \text{ am Gustave}) = P(I)P(am|I)P(Gustave|I \text{ am})$$
How to estimate these probabilities?

Can we just count and divide?

\[
P(\text{processing}|\text{I am Gustave, I love natural language}) = \frac{\text{Count}(\text{I am Gustave, I love natural language processing})}{\text{Count}(\text{I am Gustave, I love natural language})}
\]
How to estimate these probabilities?

Can we just count and divide?

\[
P(\text{processing}|\text{I am Gustave, I love natural language}) = \frac{\text{Count(\text{I am Gustave, I love natural language processing})}}{\text{Count(\text{I am Gustave, I love natural language})}}
\]

→ We’ll never see enough data for estimating long sentences
N-grams are Markov models

**Markov assumption** uses a limited context window to approximate 
\[ P(\text{processing}|\text{I am Gustave, I love natural language}) \]
N-grams are Markov models

**Markov assumption** uses a limited context window to approximate $P(\text{processing}|\text{I am Gustave, I love natural language})$

$P(\text{processing})$  *Unigram*

$P(\text{processing}|\text{language})$  *Bigram*

$P(\text{processing}|\text{natural language})$  *Trigram*
N-grams are Markov models

**Markov assumption** uses a limited context window to approximate
\[ P(\text{processing}|\text{I am Gustave, I love natural language}) \]

\[ P(\text{processing}) \quad \text{Unigram} \]

\[ P(\text{processing}|\text{language}) \quad \text{Bigram} \]

\[ P(\text{processing}|\text{natural language}) \quad \text{Trigram} \]

→ Language has long-distance dependencies, therefore n-grams are insufficient models of language
Example: estimating bigram probabilities

Estimation using $P(w_i|w_{i-1}) = \frac{\text{count}(w_i,w_{i-1})}{\text{count}(w_{i-1})}$
Example: estimating bigram probabilities

Estimation using \( P(w_i|w_{i-1}) = \frac{\text{count}(w_i, w_{i-1})}{\text{count}(w_{i-1})} \)

<s> I am Gustave </s>
<s> Gustave I am </s>
<s> I love natural language processing </s>

\[ P(\text{am}|\text{l}) = \frac{\text{count(\text{am}, \text{l})}}{\text{count(\text{l})}} = \frac{2}{3} \]
How to evaluate performance?

We calculate probabilities on a training set and evaluate on the unseen test set.
How to evaluate performance?

We calculate probabilities on a training set and evaluate on the unseen test set

We want a language model (LM) that best predicts the test set
How to evaluate performance?

We calculate probabilities on a training set and evaluate on the unseen test set.
We want a language model (LM) that best predicts the test set.
Therefore, a good LM assigns a higher probability to the test set than another LM.
If the test set has $n$ tokens, then $P(\text{test set}) = P(w_1, w_2, ..., w_n)$

\[ P_{\text{good LM}}(\text{test set}) > P_{\text{bad LM}}(\text{test set}) \]
Perplexity

Probability depends on the number of tokens, the longer the text, the smaller the probability
Perplexity

Probability depends on the number of tokens, the longer the text, the smaller the probability

→ We normalize by the number of tokens to have a metric per token:

\[
\text{Perplexity(test set)} = P(w_1, w_2, ..., w_n)^{-\frac{1}{n}}
\]

*Perplexity* is the inverse probability of the test set, normalized by the length
Perplexity

Probability depends on the number of tokens, the longer the text, the smaller the probability.

→ We normalize by the number of tokens to have a metric per token:

\[
\text{Perplexity}(\text{test set}) = P(w_1, w_2, ..., w_n)^{-\frac{1}{n}}
\]

**Perplexity** is the inverse probability of the test set, normalized by the length.

Minimizing perplexity is the same as maximizing probability.
Practical issues

Due to **unknown words**, bigrams with zero probability drop sentence probabilities to zero and prevent us from calculating perplexity.
Due to **unknown words**, bigrams with zero probability drop sentence probabilities to zero and prevent us from calculating perplexity

→ **Add-1 smoothing** pretends we saw each word one more time than we did

\[
P(w_i | w_{i-1}) = \frac{\text{count}(w_i, w_{i-1}) + 1}{\text{count}(w_{i-1}) + V}
\]

where \( V \) is the vocabulary size
Practical issues

Due to unknown words, bigrams with zero probability drop sentence probabilities to zero and prevent us from calculating perplexity

→ Add-1 smoothing pretends we saw each word one more time than we did

\[
P(w_i|w_{i-1}) = \frac{\text{count}(w_i, w_{i-1}) + 1}{\text{count}(w_{i-1}) + V}
\]

where \( V \) is the vocabulary size

To avoid underflow, every computation is performed in log space

\[
\log(p_1 \times p_2 \times p_3) = \log(p_1) + \log(p_2) + \log(p_3)
\]
Better n-grams using backoff or interpolation methods

**Backing off** through progressively shorter context models under certain conditions. For example, use trigram if \( \text{count}(w_i, w_{i-1}, w_{i-2}) > 0 \), otherwise use bigram.
Better n-grams using backoff or interpolation methods

**Backing off** through progressively shorter context models under certain conditions. For example, use trigram if $\text{count}(w_i, w_{i-1}, w_{i-2}) > 0$, otherwise use bigram.

**Interpolation** methods train individual models for different n-gram orders and then interpolate them together.

$$\hat{P}(w_n|w_{n-2}, w_{n-1}) = \lambda_1 P(w_n|w_{n-2}, w_{n-1}) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n)$$

where $\sum_{i=1}^{3} \lambda_i = 1$
From n-gram to neural network language models

Neural network language models solve major problems with n-grams

▶ The number of parameters increases exponentially as the n-gram order increases

▶ N-grams have no way to generalize from training to test set

Neural language models instead project words into a continuous space in which words with similar contexts have similar representations
How to represent word meaning?

Word meaning as a point in a multidimensional space

**Figure:** A three-dimensional affective space of connotative meaning by Osgood et al. (1957)
How to represent word meaning?

Defining meaning by linguistic distribution

The meaning of a word is its use in a language, Ludwig Wittgenstein (1953)

If A and B have almost identical environments (words around them), then they are synonyms, Zellig Harris (1954)
How to represent word meaning?

Word meaning as a point in a multidimensional space + Defining meaning by linguistic distribution = **Defining meaning as a point in a multidimensional space based on linguistic distribution**

The meaning of a word is a vector called an **embedding**
Components of a machine learning classifier

- A feature representation of the input $x$
- A classification function that computes the estimated class
- An objective function for learning (e.g., cross-entropy loss)
- An algorithm for optimizing the objective function (e.g., stochastic gradient descent)
Feedforward neural networks

Gustave Cortal
Binary logistic regression

Binary Logistic Regression
\[ p(+) = 1 - p(-) \]

Output sigmoid

Weight vector

Input feature vector

Input words

\[ \hat{y} \]

\[ W \]

\[ [1 \times f] \]

\[ X \]

\[ [f \times 1] \]

- wordcount = 3
- positive lexicon words = 1
- count of “no” = 0

dessert was great
Multinomial logistic regression

Multinomial Logistic Regression

\[ p(+) \quad p(-) \quad p(\text{neut}) \]

\[ \hat{y}_1 \quad \hat{y}_2 \quad \hat{y}_3 \]

Output
softmax

Y
[K x 1]

Weight
matrix

W
[K x f]

Input feature
vector

X
[f x 1]

\[ x_1 \quad x_2 \quad x_3 \quad \ldots \quad x_f \]

wordcount = 3
positive lexicon words = 1
count of “no” = 0

dessert was great

These \( f \) red weights are a row of \( W \) corresponding to weight vector \( w_3 \),
(= weights for class 3)
Neural unit

\[ y = \sigma(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))} \]
Activation functions (1)

\[
\sigma(z) = \frac{1}{1 + e^{-z}}
\]

**Figure:** Sigmoid function.
Activation functions (2)

\[
y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}
\]

\[
y = \text{ReLU}(z) = \max(z, 0)
\]

**Figure:** Tanh and ReLU functions.
Feedforward network (1)

\[ h = \sigma(Wx + b) \]
\[ z = Uh \]
\[ y = \text{softmax}(z) \]
Feedforward network (2)

Figure: Feedforward network sentiment analysis using traditional hand-built features.
Feedforward network sentiment analysis using a pooled embedding.

Figure: Feedforward network sentiment analysis using a pooled embedding.
Without activation functions, a multi-layer NN is equivalent to a single-layer NN

Consider the first two layers of a neural network with purely linear transformations:

\[ z^{[1]} = W^{[1]} x + b^{[1]} \]
\[ z^{[2]} = W^{[2]} z^{[1]} + b^{[2]} \]
Without activation functions, a multi-layer NN is equivalent to a single-layer NN

Consider the first two layers of a neural network with purely linear transformations:

\[ z^{[1]} = W^{[1]} x + b^{[1]} \]
\[ z^{[2]} = W^{[2]} z^{[1]} + b^{[2]} \]

The operations performed by the network can be combined and simplified as follows:

\[ z^{[2]} = W^{[2]} (W^{[1]} x + b^{[1]}) + b^{[2]} \]
\[ = W^{[2]} W^{[1]} x + W^{[2]} b^{[1]} + b^{[2]} \]
\[ = W_0 x + b_0 \]
The loss function for a single example $x$ in the context of a multi-class classification problem, with $K$ output classes, is defined as the cross-entropy loss $L_{CE}$:

$$L_{CE}(\hat{y}, y) = - \sum_{k=1}^{K} y_k \log(\hat{y}_k)$$
Loss function

The loss function for a single example $x$ in the context of a multi-class classification problem, with $K$ output classes, is defined as the cross-entropy loss $L_{CE}$:

$$L_{CE}(\hat{y}, y) = -\sum_{k=1}^{K} y_k \log(\hat{y}_k)$$

The loss $L_{CE}$ for a prediction $\hat{y}$ and true label $y$, focusing on the correct class $c$, is represented as:

$$L_{CE}(\hat{y}, y) = -\log(\hat{y}_c)$$

$$= -\log \left( \frac{\exp(z_c)}{\sum_{j=1}^{K} \exp(z_j)} \right)$$
Computing the gradients

For deep networks, computing the gradients for each weight is difficult, since we are computing the derivative with respect to weight parameters that appear all the way back in the very early layers of the network.
Computing the gradients

For deep networks, computing the gradients for each weight is difficult, since we are computing the derivative with respect to weight parameters that appear all the way back in the very early layers of the network. The solution to computing this gradient is an algorithm called error backpropagation.
Forward pass

\[ L(a, b, c) = c(a + 2b) \]
Backward pass

\[ L(a, b, c) = c(a + 2b) \]
Backpropagation calculus, 3Blue1Brown’s video

https://www.youtube.com/watch?v=Ilg3gGewQ5U
A feedforward neural language model takes as input at time $t$ a representation of some number of previous words ($w_{t-1}, w_{t-2}, \text{etc.}$) and outputs a probability distribution over possible next words.
A feedforward neural language model takes as input at time $t$ a representation of some number of previous words ($w_{t-1}, w_{t-2}$, etc.) and outputs a probability distribution over possible next words.

Like the n-gram, it approximates the probability of a word given the entire prior context by approximating based on the $n-1$ previous words:

$$P(w_t|w_1, ..., w_{t-1}) \approx P(w_t|w_{t-N+1}, ..., w_{t-1})$$
A feedforward neural language model takes as input at time $t$ a representation of some number of previous words ($w_{t-1}, w_{t-2}, \text{etc.}$) and outputs a probability distribution over possible next words.

Like the $n$-gram, it approximates the probability of a word given the entire prior context by approximating based on the $n-1$ previous words:

$$P(w_t | w_1, \ldots, w_{t-1}) \approx P(w_t | w_{t-N+1}, \ldots, w_{t-1})$$

Unlike $n$-gram models, neural language models can handle much longer histories, generalize better over contexts of similar words, and are more accurate at word prediction.
Feedforward NN for language modeling

The equations for a neural language model with a window size of 3, given one-hot input vectors for each input context word, are:

\[
e = [E_{t-3}; E_{t-2}; E_{t-1}]
\]

\[
h = \sigma(We + b)
\]

\[
z = Uh
\]

\[
\hat{y} = \text{softmax}(z)
\]
Feedforward NN for language modeling

input layer
one-hot vectors

embedding layer

hidden layer

output layer
softmax

... and thanks for all the ?

\[
\begin{align*}
X & \rightarrow W & \rightarrow E & \rightarrow h_1 & \rightarrow h_2 & \rightarrow h_3 & \rightarrow \ldots & \rightarrow Y \\
|V| \times d & \rightarrow 3d \times 1 & \rightarrow d_h \times 3d & \rightarrow d_h \times 1 & \rightarrow |V| \times d_h & \rightarrow |V| \times 1 \\
& & \downarrow & \uparrow & \downarrow & \uparrow & \\
& & 451 & \rightarrow 35 & \rightarrow 992 & \rightarrow \ldots & \rightarrow \ldots \\
& & w_t & \rightarrow w_{t-1} & \rightarrow w_{t-2} & \rightarrow w_{t-3} & \\
& & \text{for} & \rightarrow \text{all} & \rightarrow \text{the} & \rightarrow ? & \\
& & \text{and} & \rightarrow \text{thanks} & \rightarrow \text{for} & \rightarrow \text{all} & \rightarrow \text{the} & \rightarrow ?
\end{align*}
\]
Embedding

\[
\begin{array}{c}
1 \quad 5 \quad |V| \\
0000100...0000
\end{array}
\times
\begin{array}{c}
5 \\
E \\
|V|
\end{array}
= \begin{array}{c}
1 \\
d
\end{array}
\]
Embedding

\[
\begin{align*}
1 & \quad \begin{array}{c} 5 \\ |V| \end{array} \\
\begin{array}{c} 00000100...00000 \\
\end{array} & \times & \begin{array}{c} 5 \\ |V| \end{array} & \quad E & = & \begin{array}{c} 1 \\
\end{array} \\
\begin{array}{c} 00000100...00000 \\
00000000...0010 \end{array} & \times & \begin{array}{c} 5 \\ |V| \end{array} & \quad E & = & \begin{array}{c} N \\
\end{array}
\end{align*}
\]
Feedforward NN for language modeling

- **Input layer**: one-hot vectors
- **Embedding layer**: transforms input tokens into a fixed-length vector representation
- **Hidden layer**: multiple fully connected layers that learn to represent the input in a more discriminative space
- **Output layer**: softmax function to predict the probability distribution over the output vocabulary

The equation for the loss function is:

\[ L = -\log P(\text{fish} \mid \text{for, all, the}) \]
How to improve the training?

Find good **hyperparameters**: batch size, learning rate, activation functions, number of hidden layers, number of neural units.
How to improve the training?

Find good **hyperparameters**: batch size, learning rate, activation functions, number of hidden layers, number of neural units

Apply **regularization** methods: normalize input values, add dropout, add weight decay
How to improve the training?

Find good **hyperparameters**: batch size, learning rate, activation functions, number of hidden layers, number of neural units

Apply **regularization** methods: normalize input values, add dropout, add weight decay

Other techniques include label smoothing, cutting gradients norm, augmenting data, using good weights initialization, gradient descent with momentum (Adam optimizer), etc.
Exercices

Using PyTorch, you need to:

- Implement the logistic regression
- Implement a multi-layer feedforward network for text classification based on word2vec features
- Play with hyperparameters (see slide "How to improve the training?")
- Implement a multi-layer feedforward network for language modeling (optional)
- Study the skip-gram model (optional, see notebook from the last course)

The goal is to have a workable PyTorch training loop for your project!
Recurrent neural networks and attention mechanisms

Gustave Cortal
Language is a temporal phenomenon
Language is a **temporal** phenomenon

*Feedforward neural networks* assumed **simultaneous access**: for language modeling, they look only at a fixed-size window of words, then slide this window over the input.
Introduction

Language is a **temporal** phenomenon

*Feedforward neural networks* assumed **simultaneous access**: for language modeling, they look only at a fixed-size window of words, then slide this window over the input.

*Recurrent neural networks* handle the temporal nature of language without using arbitrary fixed-sized windows: the hidden layer from the previous step provides a **memory** that encodes earlier processing and informs the decisions to be made at later steps.
Feedforward vs recurrent neural networks

(a) Feedforward neural network

(b) Recurrent neural network
Recurrent neural networks

\[ h_t = g(Uh_{t-1} + Wx_t) \]

\[ y_t = \text{softmax}(Vh_t) \]
Recurrent neural networks
RNNs for language modeling

\[ \frac{1}{T} \sum_{i=1}^{T} L_{CE} \]
RNNs for other tasks

a) sequence labeling

b) sequence classification

c) language modeling

d) encoder-decoder
Stacked RNNs

\[ y_1 \rightarrow RNN_{1} \rightarrow x_1 \]
\[ y_2 \rightarrow RNN_{2} \rightarrow x_2 \]
\[ y_3 \rightarrow RNN_{3} \rightarrow x_3 \]
\[ \vdots \]
\[ y_n \rightarrow RNN_{n} \rightarrow x_n \]
Bidirectional RNNs
Training with encoder-decoder networks

Total loss is the average cross-entropy loss per target word:

\[ L = \frac{1}{T} \sum_{i=1}^{T} L_i \]

Decoder

- \( L_1 = -\log P(y_1) \)
- \( L_2 = -\log P(y_2) \)
- \( L_3 = -\log P(y_3) \)
- \( L_4 = -\log P(y_4) \)
- \( L_5 = -\log P(y_5) \)

Encoder

- \( x_1 \)
- \( x_2 \)
- \( x_3 \)
- \( x_4 \)

Gold answers:
- \( \text{llevó} \)
- \( \text{la} \)
- \( \text{bruja} \)
- \( \text{verde} \)
- \( \text{</s>} \)

Per-word loss

Softmax

Hidden layer(s)

Embedding layer

Input:
- \( \text{the} \)
- \( \text{green} \)
- \( \text{witch} \)
- \( \text{arrived} \)

Output:
- \( \text{la} \)
- \( \text{bruja} \)
- \( \text{verde} \)
Inference with encoder-decoder networks
Final hidden state as a fixed context vector for the decoder
The final hidden state acts as a bottleneck

This final hidden state must represent everything about the meaning of the source text
The final hidden state acts as a bottleneck

This final hidden state must represent everything about the meaning of the source text.

However, information at the beginning of the sentence may not be equally well represented in the context vector.
The attention mechanism is a solution to the bottleneck problem: it allows the decoder to get information from all the hidden states of the encoder.
Attention mechanisms: introduction

The attention mechanism is a solution to the bottleneck problem: it allows the decoder to get information from all the hidden states of the encoder.

The idea of attention is to create the single fixed-length vector $c$ by taking a weighted sum of all the encoder hidden states. The weights focus on a particular part of the source text that is relevant to the token the decoder is currently producing.
The attention mechanism is a **solution to the bottleneck problem**: it allows the decoder to get information from all the hidden states of the encoder.

The idea of attention is to create the single fixed-length vector $c$ by taking a **weighted sum of all the encoder hidden states**. The weights focus on a particular part of the source text that is relevant to the token the decoder is currently producing.

Attention thus replaces the static context vector with one that is **dynamically** derived from the encoder hidden states, **different** for each token in...
Dot-product attention

The first step in computing $c_i$ is to compute how relevant each encoder state is to the decoder state captured in $h_{i-1}^d$.
Dot-product attention

The first step in computing $c_i$ is to compute how relevant each encoder state is to the decoder state captured in $h_{i-1}^d$

Then, implement relevance as **dot-product similarity**:

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d \cdot h_j^e$$
Dot-product attention

The first step in computing $c_i$ is to compute how relevant each encoder state is to the decoder state captured in $h_{i-1}^d$

Then, implement relevance as **dot-product similarity**:

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d \cdot h_j^e$$

Then, apply a softmax to create a **vector of weights**, $\alpha_{ij}$, that tells the proportional relevance of each encoder hidden state $j$ to the prior hidden decoder state, $h_{i-1}^d$:

$$\alpha_{ij} = \frac{\exp(\text{score}(h_{i-1}^d, h_j^e))}{\sum_k \exp(\text{score}(h_{i-1}^d, h_k^e))}$$
Dot-product attention

The first step in computing $c_i$ is to compute how relevant each encoder state is to the decoder state captured in $h_{i-1}^d$.

Then, implement relevance as **dot-product similarity**:

$$\text{score}(h_{i-1}^d, h_j^e) = h_{i-1}^d \cdot h_j^e$$

Then, apply a softmax to create a vector of weights, $\alpha_{ij}$, that tells the proportional relevance of each encoder hidden state $j$ to the prior hidden decoder state, $h_{i-1}^d$:

$$\alpha_{ij} = \frac{\exp(\text{score}(h_{i-1}^d, h_j^e))}{\sum_k \exp(\text{score}(h_{i-1}^d, h_k^e))}$$

Finally, compute a fixed-length context vector for the current decoder state by taking a weighted average over all the encoder hidden states:

$$c_i = \sum_j \alpha_{ij} h_j^e$$
Encoder-decoder networks with dot-product attention
Long Short Term Memory network

LSTM
FNN vs RNN vs LSTM units

(a) FNN

(b) RNN

(c) LSTM Unit
Transformers

Gustave Cortal
Transformers vs recurrent neural networks

The transformer offers new mechanisms (**positional encodings** and **self-attention**) that help represent time and help focus on how words relate to each other over long distances.
Transformers vs recurrent neural networks

The transformer offers new mechanisms (positional encodings and self-attention) that help represent time and help focus on how words relate to each other over long distances.

Unlike RNNs, the computations at each time step are independent of all the other steps and, therefore, can be performed in parallel.
Transformer block

Transformer Block

Residual connection

Layer Normalize

Feedforward

Residual connection

MultiHead Attention

\[ h_1, h_2, h_3, \ldots, h_n \]

\[ x_1, x_2, x_3, \ldots, x_n \]
Self-attention layer

Self-attention directly extracts and uses information from arbitrarily large contexts without passing it through intermediate recurrent connections.
Self-attention layer

Self-attention directly extracts and uses information from arbitrarily large contexts without passing it through intermediate recurrent connections.
Attention visualization

Layer 6

Layer 5

self-attention distribution
Main idea of attention mechanisms

An attention-based approach is a set of comparisons to relevant items in some context, a normalization of those scores to provide a probability distribution, and a weighted sum using this distribution.
Dot-product attention

A dot product is the simplest form of comparison between elements in a self-attention layer:

\[ \text{score}(x_i, x_j) = x_i \cdot x_j \]
Dot-product attention

A dot product is the simplest form of comparison between elements in a self-attention layer:

$$\text{score}(x_i, x_j) = x_i \cdot x_j$$

Then, we normalize the scores with a softmax to create a vector of weights, $$\alpha_{ij}$$, that indicates the proportional relevance of each input $$j$$ to the input element $$i$$

Finally, we generate an output value $$y_i$$ by taking the sum of the inputs seen so far, weighted by their respective $$\alpha$$ value.

$$y_i = \sum_{j \leq i} \alpha_{ij} x_j$$
Dot-product attention

A **dot product** is the simplest form of comparison between elements in a self-attention layer:

\[
score(x_i, x_j) = x_i \cdot x_j
\]

Then, we **normalize** the scores with a softmax to create a vector of weights, \( \alpha_{ij} \), that indicates the proportional relevance of each input \( j \) to the input element \( i \)

\[
\alpha_{ij} = \text{softmax}(score(x_i, x_j)) \quad \forall j \leq i
\]

\[
= \frac{\exp(score(x_i, x_j))}{\sum_{k=1}^{i} \exp(score(x_i, x_k))} \quad \forall j \leq i
\]
Dot-product attention

A **dot product** is the simplest form of comparison between elements in a self-attention layer:

$$\text{score}(x_i, x_j) = x_i \cdot x_j$$

Then, we **normalize** the scores with a softmax to create a vector of weights, $\alpha_{ij}$, that indicates the proportional relevance of each input $j$ to the input element $i$

$$\alpha_{ij} = \text{softmax}(\text{score}(x_i, x_j)) \quad \forall j \leq i$$

$$= \frac{\exp(\text{score}(x_i, x_j))}{\sum_{k=1}^{i} \exp(\text{score}(x_i, x_k))} \quad \forall j \leq i$$

Finally, we generate an output value $y_i$ by taking the **sum** of the inputs seen so far, **weighted** by their respective $\alpha$ value.

$$y_i = \sum_{j \leq i} \alpha_{ij} x_j$$
Attention with queries, keys and values

But transformers create a more sophisticated way of representing how words can contribute to the representation of longer inputs. Consider the three roles each input embedding plays during the attention process:

- As the current focus of attention when being compared to all of the other preceding inputs \(\rightarrow\) query
- In its role as a preceding input being compared to the current focus of attention \(\rightarrow\) key
- And finally, as a value used to compute the output for the current focus of attention
Attention with queries, keys and values

But transformers create a more sophisticated way of representing how words can contribute to the representation of longer inputs. Consider the three roles each input embedding plays during the attention process:

- As the current focus of attention when being compared to all of the other preceding inputs \( \rightarrow \) query
- In its role as a preceding input being compared to the current focus of attention \( \rightarrow \) key
- And finally, as a value used to compute the output for the current focus of attention

To capture these three different roles, transformers introduce weight matrices \( W_Q, W_K, \) and \( W_V \). These weights project each input vector \( x_i \) into a representation of its role as a key, query, or value:

\[
q_i = W_Q x_i, \\
k_i = W_K x_i, \\
v_i = W_V x_i
\]

\( x_i \in \mathbb{R}^{d \times 1}, W_Q \in \mathbb{R}^{d \times d}, W_K \in \mathbb{R}^{d \times d}, \) and \( W_V \in \mathbb{R}^{d \times d} \).
Attention with queries, keys and values

Given these projections, the score between a current focus of attention, $x_i$, and an element in the preceding context, $x_j$, consists of a dot product between its query vector $q_i$ and the preceding element’s key vectors $k_j$:

$$\text{score}(x_i, x_j) = q_i \cdot k_j$$
Attention with queries, keys and values

Given these projections, the score between a current focus of attention, \( x_i \), and an element in the preceding context, \( x_j \), consists of a dot product between its query vector \( q_i \) and the preceding element’s key vectors \( k_j \):

\[
\text{score}(x_i, x_j) = q_i \cdot k_j
\]

The output calculation for \( y_i \) is now based on a weighted sum over the value vectors \( v \):

\[
y_i = \sum_{j \leq i} \alpha_{ij} v_j
\]
Attention with queries, keys and values

Given these projections, the score between a current focus of attention, $x_i$, and an element in the preceding context, $x_j$, consists of a dot product between its query vector $q_i$ and the preceding element’s key vectors $k_j$:

$$\text{score}(x_i, x_j) = q_i \cdot k_j$$

The output calculation for $y_i$ is now based on a weighted sum over the value vectors $v$:

$$y_i = \sum_{j \leq i} \alpha_{ij} v_j$$

Exponentiating large values can lead to numerical issues. To avoid this, we **scale** the dot-product by a factor related to the size of the embeddings:

$$\text{score}(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d}}$$
Attention with queries, keys and values

1. Generate key, query, value vectors

2. Compare x3’s query with the keys for x1, x2, and x3

3. Divide score by $d_k$

4. Turn into weights via softmax

5. Weigh each value vector $\alpha_{i,j}$

6. Sum the weighted value vectors

Output of self-attention $a_3$
Parallelization

Since each output $y_i$ is computed independently, the entire process can be parallelized by taking advantage of matrix multiplication.
Parallelization

Since each output \( y_i \) is computed independently, the entire process can be parallelized by taking advantage of matrix multiplication.

Input tokens are packed into a single matrix \( X \in \mathbb{R}^{N \times d} \). We multiply \( X \) by the key, query, and value matrices:

\[
Q = XW_Q; \quad K = XW_K; \quad V = XW_V
\]

\( Q \in \mathbb{R}^{N \times d}, \ K \in \mathbb{R}^{N \times d}, \ \text{and} \ V \in \mathbb{R}^{N \times d} \)
Parallelization

Since each output $y_i$ is computed independently, the entire process can be parallelized by taking advantage of matrix multiplication.

Input tokens are packed into a single matrix $X \in \mathbb{R}^{N \times d}$. We multiply $X$ by the key, query, and value matrices:

$$Q = XW_Q; \quad K = XW_K; \quad V = XW_V$$

$Q \in \mathbb{R}^{N \times d}$, $K \in \mathbb{R}^{N \times d}$, and $V \in \mathbb{R}^{N \times d}$

We’ve reduced the self-attention step for a sequence of $N$ tokens:

$$\text{SelfAttention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d}} \right) V$$
**Masked attention matrix**

\[ QK^T \] results in a score for each query value to every key value, including those that follow the query.
Masked attention matrix

\[ QK^T \] results in a score for each query value to every key value, including those that follow the query.

This is inappropriate in language modeling since guessing the next word is pretty simple if you already know it. To fix this, the elements in the upper-triangular portion of the matrix are set to \(-\infty\).
Transformer block
Multihead attention

Different words in a sentence can relate to each other in many different ways simultaneously.
Multihead attention

Different words in a sentence can relate to each other in many different ways simultaneously.

It is difficult for a transformer block to capture all kinds of parallel relations among its inputs.
Multihead attention

Different words in a sentence can relate to each other in many different ways simultaneously.

It is difficult for a transformer block to capture all kinds of parallel relations among its inputs.

Transformers address this issue with multihead self-attention layers, sets of self-attention layers, called heads, that reside in parallel layers at the same depth in a model, each with its own set of parameters.
Multihead attention

Different words in a sentence can relate to each other in many different ways simultaneously.

It is difficult for a transformer block to capture all kinds of parallel relations among its inputs.

Transformers address this issue with **multihead self-attention layers**, sets of self-attention layers, called heads, that reside in parallel layers at the same depth in a model, each with its own set of parameters.

Given these distinct sets of parameters, each head can learn different aspects of the relationships among inputs at the same level of abstraction.
Multihead attention

Multihead Attention Layer with h=4 heads

Project from $hd_v$ to d

Concatenate Outputs $[N \times hd_v]$

$[N \times d]$

$[N \times d]$

$w_Q^1, w_K^1, w_V^1$  Head 1

$w_Q^2, w_K^2, w_V^2$  Head 2

$w_Q^3, w_K^3, w_V^3$  Head 3

$w_Q^4, w_K^4, w_V^4$  Head 4

$w^O$ $[hd_v \times d]$
Multihead attention

To implement this notion, each head, \( i \), in a self-attention layer is provided with its own set of key, query, and value matrices: \( W^K_i \), \( W^Q_i \), and \( W^V_i \).
Multihead attention

To implement this notion, each head, $i$, in a self-attention layer is provided with its own set of key, query, and value matrices: $W^K_i$, $W^Q_i$, and $W^V_i$

In multi-head attention, instead of using the model dimension $d$ that’s used for the input and output from the model, the key and query embeddings have dimensionality $d_k << d$
Multihead attention

To implement this notion, each head, $i$, in a self-attention layer is provided with its own set of key, query, and value matrices: $W^K_i$, $W^Q_i$, and $W^V_i$

In multi-head attention, instead of using the model dimension $d$ that’s used for the input and output from the model, the key and query embeddings have dimensionality $d_k << d$

\[
\text{MultiHeadAttention}(X) = (\text{head}_1 \oplus \text{head}_2 \ldots \oplus \text{head}_h)W^O
\]

\[
Q_i = XW^Q_i; \quad K_i = XW^K_i; \quad V_i = XW^V_i
\]

\[
\text{head}_i = \text{SelfAttention}(Q_i, K_i, V_i)
\]

$X \in \mathbb{R}^{N \times d}$

$W^Q_i \in \mathbb{R}^{d \times d_k}$, $W^K_i \in \mathbb{R}^{d \times d_k}$, and $W^V_i \in \mathbb{R}^{d \times d_v}$

$Q \in \mathbb{R}^{N \times d_k}$, $K \in \mathbb{R}^{N \times d_k}$, and $V \in \mathbb{R}^{N \times d_v}$

$W^O \in \mathbb{R}^{hd_v \times d}$
Transformer block

Transformer Block

Residual connection

Layer Normalize

Feedforward

Residual connection

MultiHead Attention
Residual connections

Residual connections pass information from a lower layer to a higher layer without going through the intermediate layer.
Residual connections

Residual connections pass information from a lower layer to a higher layer without going through the intermediate layer.

Allowing information from the activation going forward and the gradient going backward to skip a layer improves learning and gives higher-level layers direct access to information from lower layers.
Residual connections

Residual connections pass information from a lower layer to a higher layer without going through the intermediate layer.

Allowing information from the activation going forward and the gradient going backward to skip a layer improves learning and gives higher-level layers direct access to information from lower layers.

If we think of a layer as one long vector of units, the resulting function computed in a transformer block can be expressed as:

\[
O = \text{LayerNorm}(X + \text{SelfAttention}(X)) \\
H = \text{LayerNorm}(O + \text{FFN}(O))
\]
Transformer block
Layer normalization

\[
O = \text{LayerNorm}(X + \text{SelfAttention}(X))
\]
\[
H = \text{LayerNorm}(O + \text{FFN}(O))
\]
Layer normalization

\[ O = \text{LayerNorm}(X + \text{SelfAttention}(X)) \]

\[ H = \text{LayerNorm}(O + \text{FFN}(O)) \]

We calculate the mean, \( \mu \), and standard deviation, \( \sigma \), over the elements of the vector to be normalized. Given a hidden layer with dimensionality \( d \), these values are calculated as follows:

\[
\mu = \frac{1}{d} \sum_{i=1}^{d} x_i
\]

\[
\sigma = \sqrt{\frac{1}{d} \sum_{i=1}^{d} (x_i - \mu)^2}
\]

\[
\hat{x} = \frac{(x - \mu)}{\sigma}
\]
Train positional embeddings or use a static function that maps integer inputs to real-values vectors
**Language Model Head**

- Takes $h^L_N$ and outputs a distribution over vocabulary $V$.

---

**Layer L Transformer Block**

- $h^L_1$, $h^L_2$, ..., $h^L_N$
- $1 \times d$

---

**Unembedding layer**

- $E^T$
- $d \times |V|$

---

**Softmax over vocabulary $V$**

- Logits $1 \times |V|$
- Word probabilities $1 \times |V|$
Language modeling using next word prediction

\[
\frac{1}{T} \sum_{t=1}^{T} L_{CE}
\]
Conditional generation
Causal vs bidirectional language model

a) A causal self-attention layer

b) A bidirectional self-attention layer
Attention matrix for bidirectional language model

<table>
<thead>
<tr>
<th></th>
<th>q1•k1</th>
<th>q1•k2</th>
<th>q1•k3</th>
<th>q1•k4</th>
<th>q1•k5</th>
</tr>
</thead>
<tbody>
<tr>
<td>q2•k1</td>
<td>q2•k1</td>
<td>q2•k2</td>
<td>q2•k3</td>
<td>q2•k4</td>
<td>q2•k5</td>
</tr>
<tr>
<td>q3•k1</td>
<td>q3•k2</td>
<td>q3•k3</td>
<td>q3•k4</td>
<td>q3•k5</td>
<td></td>
</tr>
<tr>
<td>q4•k1</td>
<td>q4•k2</td>
<td>q4•k3</td>
<td>q4•k4</td>
<td>q4•k5</td>
<td></td>
</tr>
<tr>
<td>q5•k1</td>
<td>q5•k2</td>
<td>q5•k3</td>
<td>q5•k4</td>
<td>q5•k5</td>
<td></td>
</tr>
</tbody>
</table>

N

N
Masked language modeling

CE Loss

Softmax over Vocabulary

Token + Positional Embeddings

Bidirectional Transformer Encoder
Sequence classification

**Diagram:**

- **Input:** [CLS], entirely, predictable, and, lacks, energy
- **Encoder:** Bidirectional Transformer Encoder
- **Output:** $y_{CLS}$
- **Softmax:** $z_{CLS}$

**Legend:**

- Word + Positional Embeddings
Token classification

$$\text{argmax} \quad \text{NNP}$$

$$\text{softmax} \quad \text{MD}$$

$$W_K \quad \text{VB}$$

$$z_i \quad \text{DT}$$

$$y_i \quad \text{NN}$$

Bidirectional Transformer Encoder

Embedding Layer

[CLS] Janet will back the bill
Transformer architecture from *Attention is All you Need*
### Architecture, size, and hyperparameters of GPT-3 from *Language Models are Few-Shot Learners*

<table>
<thead>
<tr>
<th>Model Name</th>
<th>$n_{\text{params}}$</th>
<th>$n_{\text{layers}}$</th>
<th>$d_{\text{model}}$</th>
<th>$n_{\text{heads}}$</th>
<th>$d_{\text{head}}$</th>
<th>Batch Size</th>
<th>Learning Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT-3 Small</td>
<td>125M</td>
<td>12</td>
<td>768</td>
<td>12</td>
<td>64</td>
<td>0.5M</td>
<td>$6.0 \times 10^{-4}$</td>
</tr>
<tr>
<td>GPT-3 Medium</td>
<td>350M</td>
<td>24</td>
<td>1024</td>
<td>16</td>
<td>64</td>
<td>0.5M</td>
<td>$3.0 \times 10^{-4}$</td>
</tr>
<tr>
<td>GPT-3 Large</td>
<td>760M</td>
<td>24</td>
<td>1536</td>
<td>16</td>
<td>96</td>
<td>0.5M</td>
<td>$2.5 \times 10^{-4}$</td>
</tr>
<tr>
<td>GPT-3 XL</td>
<td>1.3B</td>
<td>24</td>
<td>2048</td>
<td>24</td>
<td>128</td>
<td>1M</td>
<td>$2.0 \times 10^{-4}$</td>
</tr>
<tr>
<td>GPT-3 2.7B</td>
<td>2.7B</td>
<td>32</td>
<td>2560</td>
<td>32</td>
<td>80</td>
<td>1M</td>
<td>$1.6 \times 10^{-4}$</td>
</tr>
<tr>
<td>GPT-3 6.7B</td>
<td>6.7B</td>
<td>32</td>
<td>4096</td>
<td>32</td>
<td>128</td>
<td>2M</td>
<td>$1.2 \times 10^{-4}$</td>
</tr>
<tr>
<td>GPT-3 13B</td>
<td>13.0B</td>
<td>40</td>
<td>5140</td>
<td>40</td>
<td>128</td>
<td>2M</td>
<td>$1.0 \times 10^{-4}$</td>
</tr>
<tr>
<td>GPT-3 175B or “GPT-3”</td>
<td>175.0B</td>
<td>96</td>
<td>12288</td>
<td>96</td>
<td>128</td>
<td>3.2M</td>
<td>$0.6 \times 10^{-4}$</td>
</tr>
</tbody>
</table>
Conclusion

Tokenization is splitting text into individual tokens
Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word
Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

N-grams are simple probabilistic language models based on Markov assumption
Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

N-grams are simple probabilistic language models based on Markov assumption

Naive bayes classifiers are generative models based on class-specific unigram
Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

N-grams are simple probabilistic language models based on Markov assumption

Naive bayes classifiers are generative models based on class-specific unigram

Embedding represents word meaning as a vector
Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

N-grams are simple probabilistic language models based on Markov assumption

Naive bayes classifiers are generative models based on class-specific unigram

Embedding represents word meaning as a vector

Logistic regressions are discriminative models based on the sigmoid function
Conclusion

Tokenization is splitting text into individual tokens

A language model is a probabilistic model that can compute the probability of a sequence of words and compute the probability of an upcoming word

N-grams are simple probabilistic language models based on Markov assumption

Naive bayes classifiers are generative models based on class-specific unigram

Embedding represents word meaning as a vector

Logistic regressions are discriminative models based on the sigmoid function

Feedforward neural networks handle longer inputs and generalize better compared to N-grams thanks to embeddings, have fixed context windows
Conclusion

Recurrent neural networks handle temporal data inherently in the architecture, have infinite context windows, hidden states have local information.
Conclusion

Recurrent neural networks handle temporal data inherently in the architecture, have infinite context windows, hidden states have local information

Information flow is better in gated recurrent networks due to better context management
Conclusion

Recurrent neural networks handle temporal data inherently in the architecture, have infinite context windows, hidden states have local information.

Information flow is better in gated recurrent networks due to better context management.

Attention mechanisms solve the bottleneck problem to produce dynamically derived context vectors.
Recurrent neural networks handle temporal data inherently in the architecture, have infinite context windows, hidden states have local information.

Information flow is better in gated recurrent networks due to better context management.

Attention mechanisms solve the bottleneck problem to produce dynamically derived context vectors.

Transformers use self-attention layers combined with feedforward layers to handle more complex distant relationships between tokens, enable parallelization due to independent computation between tokens, have fixed context windows.
Ressources


3Blue1Brown’s videos on *neural networks*. https://www.youtube.com/playlist?list=PLZHQObQbOWTQDNU6R1_67000Dx_ZCJB-3pi


Ressources

Harvard NLP. *The Annotated Transformer.*
https://nlp.seas.harvard.edu/annotated-transformer/

Peter Bloem. *Transformers from scratch.*
https://peterbloem.nl/blog/transfomers

Andrej Karpathy. *Let’s build GPT: From scratch, in code, spelled out.*
https://www.youtube.com/watch?v=kCc8FmEb1nY


Ressources

Collège de France, « Apprendre les langues aux machines »:
https://www.college-de-france.fr/fr/agenda/cours/apprendre-les-langues-aux-machines

Dan Jurafsky and James H. Martin, *Speech and Language Processing*:

3Blue1Brown, *Essence of linear algebra* and *Neural Networks* playlists:
https://www.youtube.com/@3blue1brown/playlists

AI News: we summarize top AI discords + AI reddits + AI X/Twitters, and send you a roundup each day!
https://buttondown.email/ainews