CSC412
Probabilistic Learning and Reasoning
Week 10 Embeddings & Attention

Denys Linkov

University of Toronto

Overview

- Recap of NLP Tasks
- Intro to Embeddings
- Intro to Attention
- Goal is to catch you up on LLMs for next two lectures

NLP Tasks

- Classification (Documents, Spans, Tokens)
  - Hate speech detection
  - Spam filtering
  - Social Media drug adverse effect identification
- Generation (Question Answering, Summarization, Free-text generation, . . .)
  - Translating natural language into SQL queries
  - "Hey siri what is the weather like?"
  - Chatbots
  - Talk to a transformer

NLP Tasks

- Regression (Essay scoring, like count prediction)
  - How many retweets will this tweet get?
- Information Extraction
  - Who are the people mentioned in this text?
  - What date was the procedure performed?
- Document Retrieval
  - Google search
  - Automatic literature review
  - "R" part of RAG

NLP concepts

Some names we will be using throughout this lecture:

- **Token** - A single "atom" of text, usually a word
- **Document** - A complete datapoint of text
- **Span** - A subset of a document, a group of Tokens
- **Vocabulary** - A list of all possible tokens

Spam

We begin by revisiting a familiar example from earlier lectures: determining whether an email is spam or not.

- What kind of task is it ?
- How would you approach it?

More formally: Consider a set of observations $(x_i, y_i)_{i=1:N}$ where $y_i = 1$ means datapoint i was spam, and x_i is the text of the email.
Our goal is to create / learn a function f_θ such that $f_\theta(x_i) = p(y_i|x_i)$

Heuristics

Perhaps it is possible to find some heuristics that work:

- If x_i contains any prescription medication name $\hat{y}_i = 1$
- If x_i is is mostly capital letters $\hat{y}_i = 1$
- If x_i contains 'Make **Amount** every **Time Period**" $\hat{y}_i = 1$

```python
import re

SPAM = 1
MEDICINE_LIST = ["Acetylcysteine", "Apixaban","..."]

def spam_classifier_heuristic(text: str) -> bool:
    """
    Heuristics to classify whether a string is a spam email or not
    """
    for medicine in MEDICINE_LIST:
        if text.contains(medicine):
            return SPAM
    if text.upper() == text:
        return SPAM
    if re.match(r"((M|m)ake)\s*((\w*)\s*)*(\$\d*)\s*((per|for\sjust)\s*(week|day|year))|(weekly|daily|monthly)", text) is not None:
        return SPAM
    else:
        return 1 - SPAM
```

Can we learn simple heuristics?

To apply any kind of learning algorithms we have seen before we need to convert x into a numerical representation h.

- Given a vocabulary $V_{j=1:M}$, determine h such that: $h_j = 1$ whenever token j from the vocabulary is present in x.
- Each datapoint x is represented by an M dimensional vector of 0's and 1's
- How do we determine the vocabulary?
- Just list and count all the words in all of the documents, and then only keep the top M.
- We have numerical features, so just plug them as input into any 'standard" algorithm: Logistic Regression

Bag of Words

This simple binary representation is called a **(binary) Bag of Words**.

- What is included in the representation h of x?
- What if we care about more than just the presence / abscence of a specific word?
- We could just include the count of each word turning h from a vector of 1's and 0's into a vector of counts.
- What about phrases? "Polyethylene Glicol"?

N-Grams

Instead of words being entries in a vocabulary, use phrases.
An **N-gram** is a contiguous sequence of N tokens from a given text.
Under $N = 1$; also called unigrams

$$V = (\text{“This”}, \text{“is”}, \text{“a”}, \text{“sentence”})$$

$$\text{“This is a sentence”} = (1, 1, 1, 1)$$

$$\text{“A sentence”} = (0, 0, 1, 1)$$

Under $N = 2$; also called bigrams

$$V = (\text{“This is”}, \text{“is a”}, \text{“a sentence”})$$

$$\text{“This is a sentence”} = (1, 1, 1)$$

$$\text{“A sentence”} = (0, 0, 1)$$

Common words

Some words are incredibly common, but do not contribute a lot in terms of distinguishing between texts.

- Virtually any text in English will contain "the" or "a"
- Should we include those in the vocabulary?
- Can we learn which words to include in the vocabulary?
- We can better represent documents by the **relative frequency** of words in them.
- Note: in practice we ~~often remove some words from all text and ignore them completely~~ throw all words in to train a LLM*. You will see those referred to as **stopwords**

Term Frequency

We can represent the "count" (**Term Frequency**) of a word in many different ways:

- Raw count of times it is present in x: **BoW**
- Binarized count of times it is present in x: **binary BoW**
- Count of times it is present in x divided by number of tokens in x
- Raw count scaled by number of other terms (not count of) in x
- Log of the raw count

Term Frequency example

$V = (\text{“This”}, \text{“is”}, \text{“a”}, \text{“sentence”}, \text{“another”}, \text{“not”}, \text{“Yet”})$
$x_1 =$“This is a sentence. This is another sentence.”
$x_2 =$“This is not a sentence. Yet another not a sentence”

Word	**TF**(w, x_1)	**TF**(w, x_2)
“This”	2	1
“is”	2	1
“a”	1	2
“sentence”	2	2
“another”	1	1
“not”	0	2
“Yet”	0	1

Table: Term Frequency (TF) for each document

Can be a raw count or a % of words in a document

Inverse Document Frequency

Just like with term frequency, we can represent the "relative prevalence" (**Inverse Document Frequency**) of a word in many different ways:
Denote $N_j = \sum_{i=1}^{N} I\left(V_j \in x_i\right)$ count of datapoints that include j-th word in the vocabulary. Denote N as the total number of documents

- $$\frac{1}{N_j}$$
- $$\frac{N}{N_j}$$
- $$\log(\frac{N}{N_j})$$
- $$\log(\frac{N}{N_j+1})$$

This is a scaling factor for how often words occur across documents

IDF Calculation

Word	**Docs with w**	**DF**(w)	**IDF**(w)
"This"	x_1, x_2	2	$\log_{10}(\frac{2}{2}) = \log_{10}(1) = 0$
"is"	x_1, x_2	2	$\log_{10}(\frac{2}{2}) = \log_{10}(1) = 0$
"a"	x_1, x_2	2	$\log_{10}(\frac{2}{2}) = \log_{10}(1) = 0$
"sentence"	x_1, x_2	2	$\log_{10}(\frac{2}{2}) = \log_{10}(1) = 0$
"another"	x_1, x_2	2	$\log_{10}(\frac{2}{2}) = \log_{10}(1) = 0$
"not"	x_2	1	$\log_{10}(\frac{2}{1}) = \log_{10}(2) \approx 0.301$
"Yet"	x_2	1	$\log_{10}(\frac{2}{1}) = \log_{10}(2) \approx 0.301$

Table: Document Frequency (DF) and Inverse Document Frequency (IDF)

TF-IDF

By combining Term Frequency with Inverse Document Frequency we can measure how common the word is in a particular datapoint relative to other documents.

$$\text{TF-IDF}(x) = TF(x) \times IDF(x)$$

Word	**TF**(w, x_2)	**IDF**(w)	**TF-IDF**(w, x_2)
"This"	1	0	$1 \times 0 = 0$
"is"	1	0	$1 \times 0 = 0$
"a"	2	0	$2 \times 0 = 0$
"sentence"	2	0	$2 \times 0 = 0$
"another"	1	0	$1 \times 0 = 0$
"not"	2	0.301	≈ 0.602
"Yet"	1	0.301	≈ 0.301

Table: TF-IDF for Document x_2

TF-IDF and BOW in Search

- Search is an important problem and one baseline is called BM25.
- Given a query Q, containing keywords $q_1, ..., q_n$, the BM25 score of a document D is:

$$\text{score}(d, q) = \sum_{w \in q} \text{IDF}(w) \cdot \frac{\text{TF}(w, d) \cdot (k_1 + 1)}{\text{TF}(w, d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{\text{avgdl}})}$$

- k_1 **(Term Frequency Saturation):**
  - Controls how quickly term frequency saturates (TF being 5, 20)
  - Typical range: [1.2, 2.0].
- b **(Document Length Normalization):**
  - Controls the degree of document length normalization.
  - $b = 1$: Full length normalization. Longer documents are penalized more heavily.
  - $b = 0$: No length normalization. Equivalent to disabling length normalization.
  - Typical value: 0.75.

Generalizable Representations

- What if we don't want to use words or skip-grams?
- How do we classify, retrieve or utalize more abstract concepts
- We can train an embedding model to map concepts into a latent space
- We did this with our Autoencoders
- Let's talk more generally about embeddings

Embeddings

All of the methods we talked about can be used to generate numerical representations of whole documents, by the use of just the word occurences, and simple functions. We say that h_i is the **embedding** of x_i, and we call $g(x) = h$ an embedding function. We can then re-frame our problem of learning $f_\theta(x) = y$ as:

$$f_\theta(x) = c_{\theta_1}(h) = c_{\theta_1}(g_{\theta_2}(x)) = y$$

Where c_θ is any classification / regression function, and g_θ is an embedding function.

- Embeddings are not restricted to documents.
- How could we embed an image?
- What if our datapoint consists of an image and text?

Why are embeddings useful?

- A function that can map text, images or other information into the same space is useful
- Compare similarity in a latent space
- Allow us to search, clustering, semi-supervised learning, etc

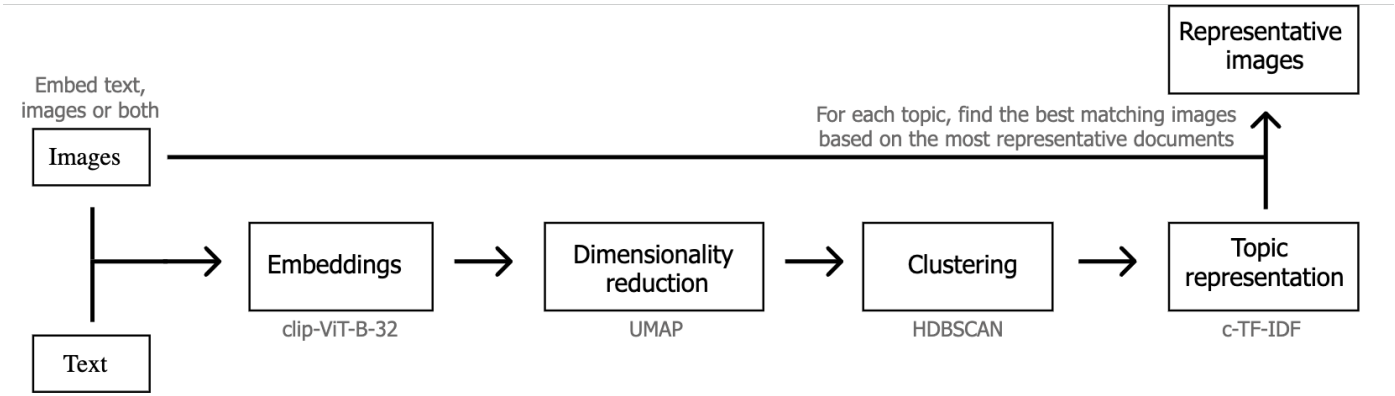


Figure: Multi Modal Clustering - Bert Topic - (Grootendorst, 2022)

Why are embeddings useful? Cont.

- Finetune models on top of embeddings
- Use transfer learning from one trained model to another
- Create a linear head or linear probe to create a mapping between the embeddings to some classes

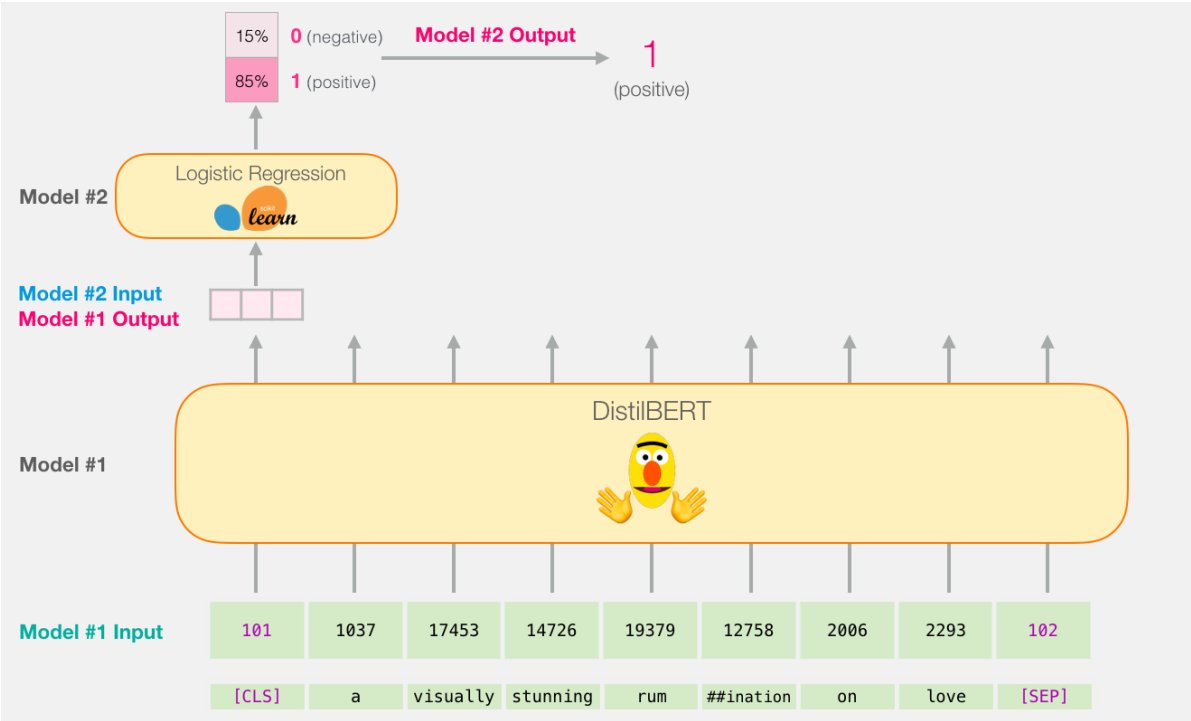
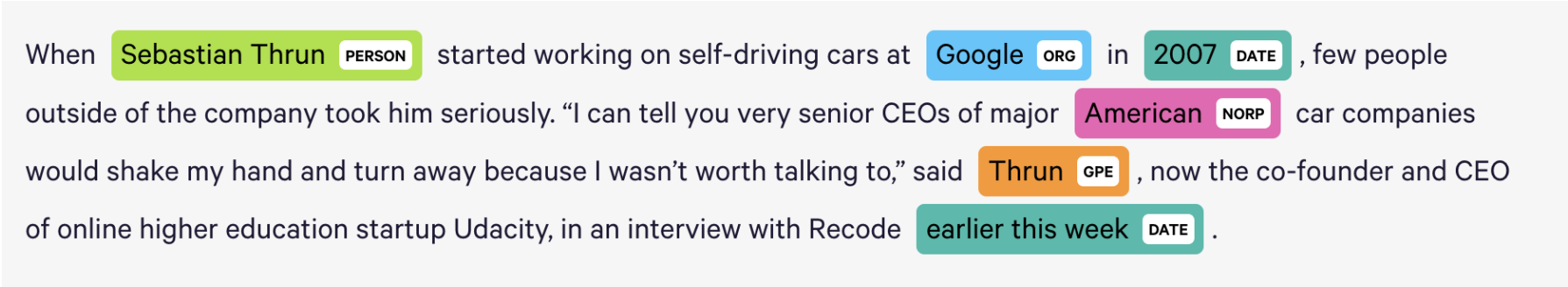


Figure: Modeling with DistilBert - (Alammar, 2018)

Token classification

Sometimes we care about something more granular than just the whole document. Perhaps we want to identify each parts of text that correspond to certain concepts:



- What should we get a represantation of?
- How could we do this?

[1] from SpaCy: https://explosion.ai/demos/displacy-ent

Word2Vec

We start with a fairly strong assumption: "Words that have similar meanings will occur in similar contexts" Based on that we define a **context** of size **k** of token $x_{i,j}$ [1]as a set of tokens:

$$\text{context}(x_{i,j}) = \{x_{i,j-k}, x_{i,j-(k-1)}, \ldots, x_{i,j-1}, x_{i,j+1}, \ldots x_{i,j+k}\}$$

Then given a set of datapoints $x_{i=1:N,j=1:M_i}$ and a vocabulary $V_{r=1:R}$ we define an unsupervised learning task of predicting what words occur in the context of each word in the vocabulary. More formally, given a sequence of training words $x_1, \ldots, x_T$ we want to maximize the average log probability:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{j\in\text{context}(t)} logp(w_j|w_t)$$

[1]This is also called a **skip-gram**

Skip Gram Continued

The basic formulation of $p(x|w)$ uses the softmax function:

$$p(x|w) = \frac{\exp\left((u(w)^T(v(x)\right)}{\sum_{r=1}^{R}\exp\left((u(w))^T(v(V_r))\right)}$$

where $u(w)$ is the "word" and $v(w)$ is the "context" representation of word w i.e the bBOW.

- In our particular case we will take u and v to be simple linear projections of the **one-hot** (binary BoW) encoding of the word, and context respectively.

$$u(w) = bBoW(w)U^T$$

The matrix U will be of size $R \times e$ where **e** is the embedding dimension, which is a hyperparameter you chose.

Skip Gram continued

Thinking about this visually we take $v(x)$ multiply it by the embedding matrix U, get a represemtation $u(w)$, multiply by U' and get our predicted context words. A nice visual below with some different notation.

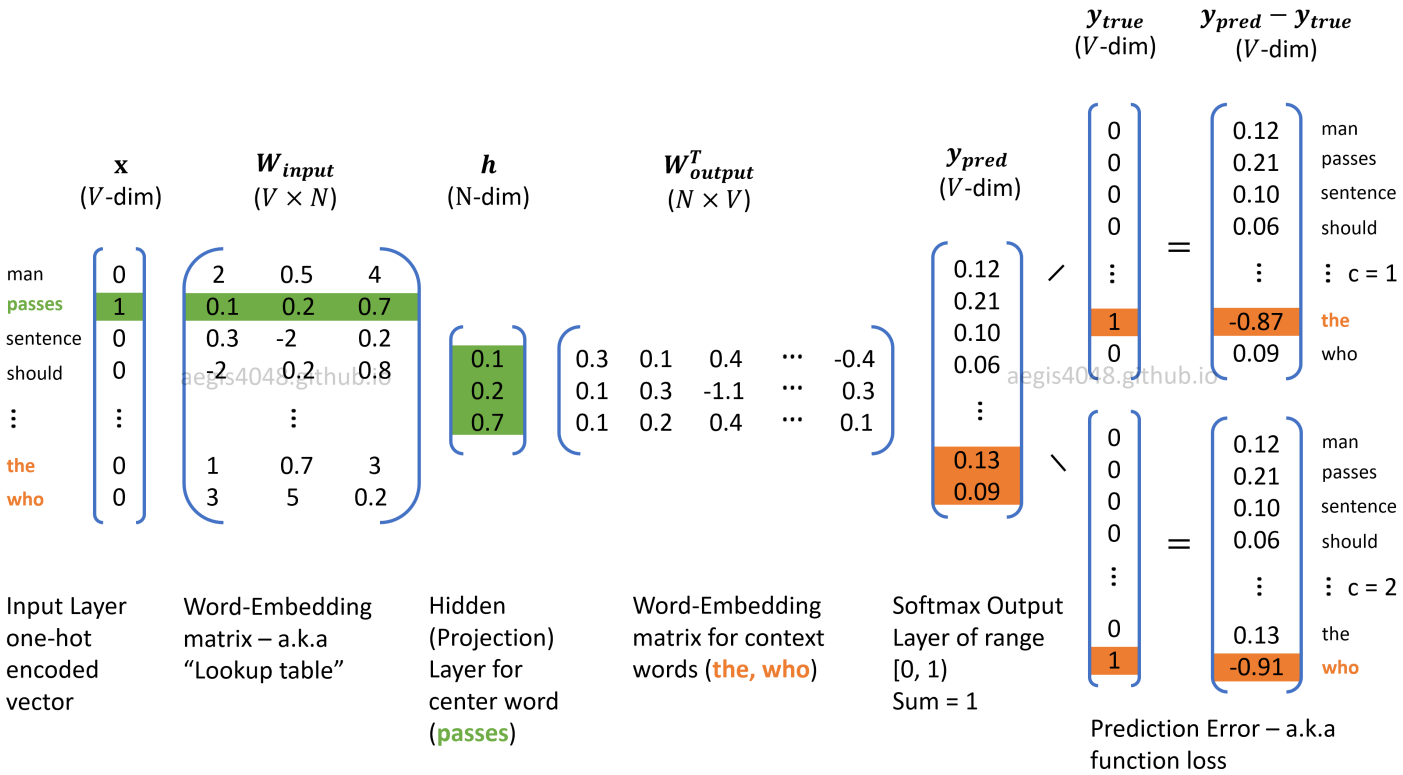
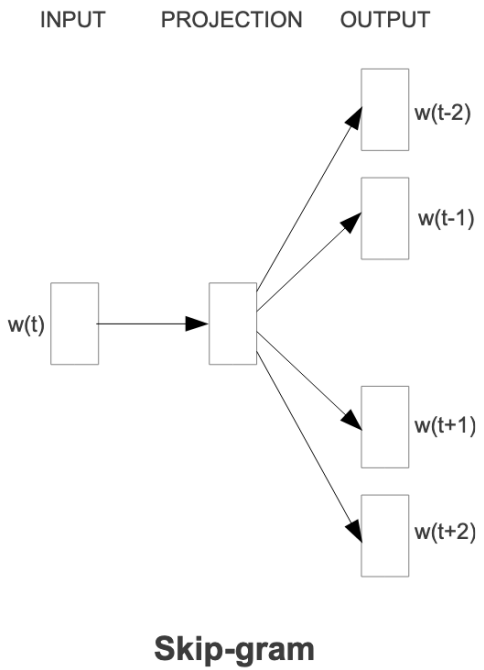


Figure: Visual example from (Pythonic Excursions, 2019)

Skip Gram Continued

Notice that the the bBoW(w) is a binary vector with all 0's and a single 1 at the index of word w in the Vocabulary!
Similarily we define $v(w)$ to be a linear projection that back from $u(w)$ to predict each word in the context.

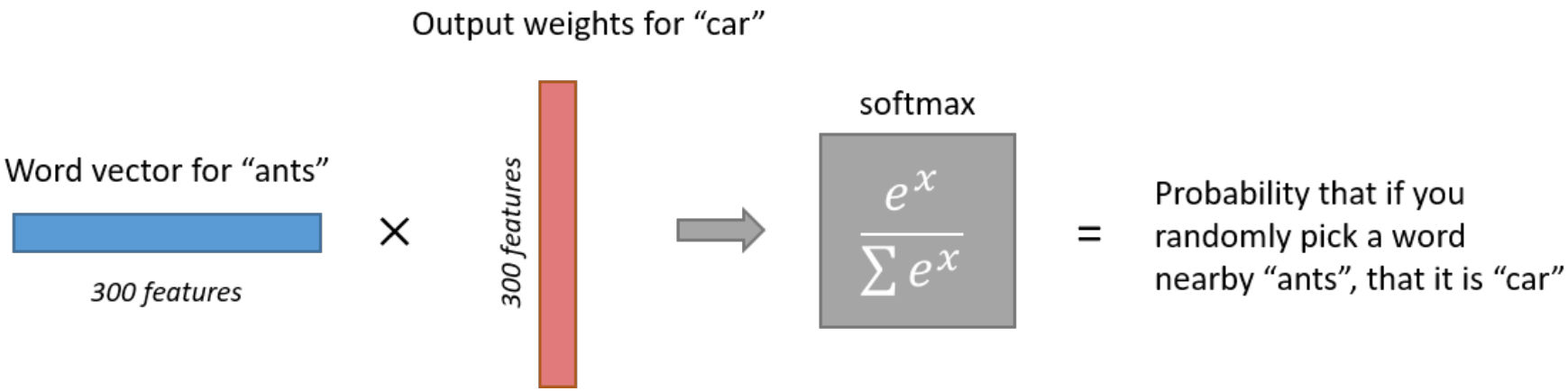


Skip-gram

[1] "Efficient Estimation of Word Representations in Vector Space", Mikolov et al

Skip Gram Vis

How to estimate $p(\text{“car”}|\text{“ants”})$?



[1] Image from: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Word2Vec

The training process is then as follows:

- Initialize the two matrices
- Sample pairs of words (w_i, w_j) and compute the objective
- Gradient Descent based on the objective.
- After convergence keep only the matrix U
- Embedding of word at vocabulary index i is just the i-th row of U

$$[0 \quad 0 \quad 0 \quad \boxed{1} \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ \boxed{10} & \boxed{12} & \boxed{19} \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

[1] Image from:
http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Word2Vec notes

- In practice this training procedure is not feasible - we would have to compute softmax over the entire vocabulary at every step.
- There are a lot of tricks and improvements over the years - really worth reading the original paper.
- There is another possible objective called **Continuous Bag of Words (CBoW)** that is the exact opposite of the Skip Gram Objective - estimate the word given context instead of context given word.

Token Classification and Embeddings continued

- To classify tokens, we can just take the Word2Vec embedding of each token as an input to e.g. Linear Regression / Multinomial Naive Bayes, and estimating probabilities that the token belongs to a certain category.
- We can combine Word2Vec representations into document level representations.
- We can combine Different embedding methods! Nothing is stopping you from taking TF-IDF vector of a document and "stacking" the average of all Word2Vec vectors in the document.

Sequence 2 Sequence Tasks

- What if our output should also be in the form of text?
- Re-frame the "context" to only feature words before the input
- Train in the unsupervised setting of CBoW, with the modified context.
- Idea: Sample the next word, conditional on the previous k based on the CBoW softmax.
- What kind of model is that?

Similarity

- What does it mean for 2 words to be similar?
- What does it mean for 2 datapoints to be similar?
- The most common way to measure similarity in NLP is via the cosine similarity

We define **cosine similarity** between two vectors to be:

$$\text{cosine sim}(x, y) = \frac{x \cdot y}{||x|| \cdot ||y||} = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}}$$

This is especially convenient for binary BoW.

Some Cool properties of W2V

Let $h(x)$ be the Word2Vec embedding of the word x. We can perform some vector algebra to find that:

- $$h(\text{“Athens”}) - h(\text{“Greece”}) + h(\text{“Germany”}) \approx h(\text{“Berlin”})$$
- $$h(\text{“Mice”}) - h(\text{“Mouse”}) + h(\text{“Dollar”}) \approx h(\text{“Dollars”})$$
- In the original paper they propose a set of 14 categories and evaluate accuracy on these kinds of “algebraic” operations to find an accuracy of $\sim 55 - 60\%$

You can just download the original Word2Vec embeddings and play around!

Modern tokenizers

- Tokenizers want to encode information into a useful way for LLMs
- Most LLMs have custom trained tokenizers, trying to make internet training efficient
- LLaMa 3 and DeepSeekv3 - 128k vocab size, GPT-4o - 200k
- Can also tokenize images

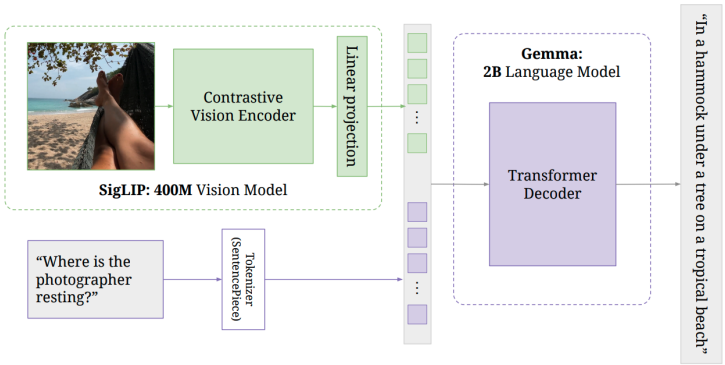


Figure: Paligemma Model - (Beyer et al, 2024)

Training a modern tokenizer

- Few approaches, find a good corpus of data
- Can focus on compression of information or representation
- GPT3, GPT4 - GPT4o reduce tokens in other languages

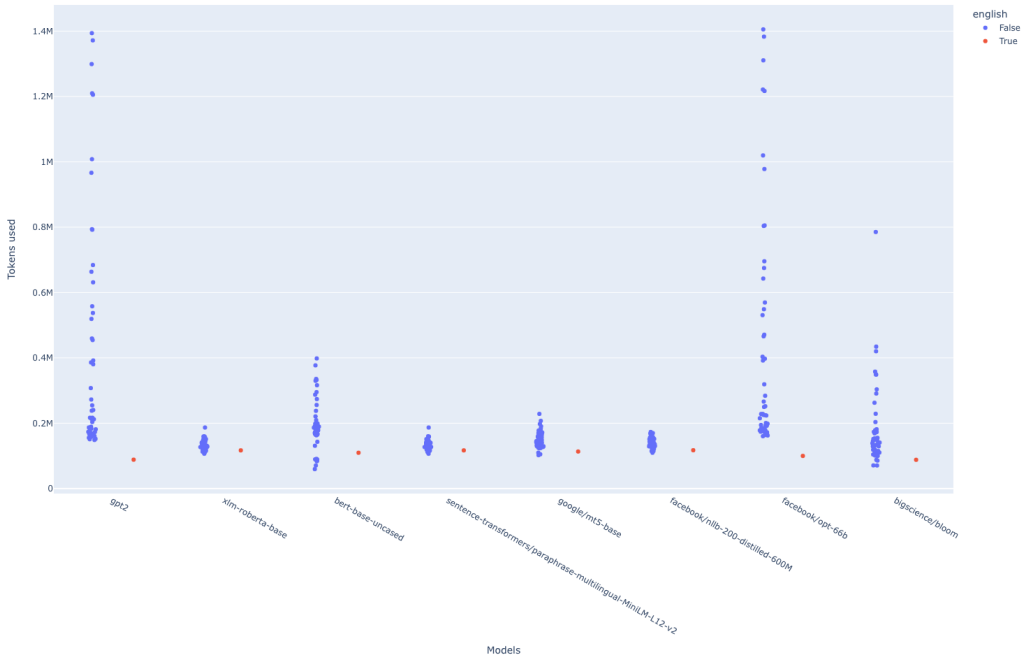


Figure: Token Comparisons Blog (Linkov, 2023)

Language tokenization

These 20 languages were chosen as representative of the new tokenizer's compression across different language families

Gujarati 4.4x fewer tokens (from 145 to 33)	હેલો, મારું નામ જીપીટી-4o છે. હું એક નવા પ્રકારનું ભાષા મોડલ છું. તમને મળીને સારું લાગ્યું!
Telugu 3.5x fewer tokens (from 159 to 45)	నమస్కారము, నా పేరు జీపీటీ-4o. నేను ఒక్క కొత్త రకమైన భాషా మోడల్ ని. మిమ్మల్ని కలిసినందుకు సంతోషం!
Tamil 3.3x fewer tokens (from 116 to 35)	வணக்கம், என் பெயர் ஜிபிடி-4o. நான் ஒரு புதிய வகை மொழி மாடல். உங்களை சந்தித்ததில் மகிழ்ச்சி!
Marathi 2.9x fewer tokens (from 96 to 33)	नमस्कार, माझे नाव जीपीटी-4o आहे! मी एक नवीन प्रकारची भाषा मॉडेल आहे! तुम्हाला भेटून आनंद झाला!
Hindi 2.9x fewer tokens (from 90 to 31)	नमस्ते, मेरा नाम जीपीटी-4o है। मैं एक नए प्रकार का भाषा मॉडल हूँ। आपसे मिलकर अच्छा लगा!
Urdu 2.5x fewer tokens (from 82 to 33)	ہیلو، میرا نام جی پی ٹی-4o ہے۔ میں ایک نئے قسم کا زبان ماڈل ہوں، آپ سے مل کر اچھا لگا!
Arabic 2.0x fewer tokens (from 53 to 26)	مرحبًا، اسمي جي بي تي-4o. أنا نوع جديد من نموذج

Figure: GPT-4o Tokenizer changes (OpenAI, 2024)

Modern embedding vector

- As we saw from previous embeddings, the representation is some kind of vector. But words a limiting.
- Let's take context into account
- Idea - Use high dimensional space to create some latent variables
- How can we use embeddings for other tasks? Attach layers afterwards

Training an embedding model - Contrastive Learning

Given a dataset X, at each training step, pick samples x_i, v_i^+, v_i^- and train an embedding function f_θ s.t.:

$$f_\theta(x) \approx f_\theta(v_i^+)$$

but not

$$f_\theta(x) \not\approx f_\theta(v_i^-)$$

Some examples:

$$L(x, v^-, v^+) = -\max(f(x)^T f(v^-) - f(x)^T f(v^+) + m, 0)$$

$$L(x, v^-, v^+) = -log\left(\frac{exp(\frac{f(x)^T f(v^+)}{\tau})}{exp(\frac{f(x)^T f(v^+)}{\tau}) + exp(\frac{f(x)^T f(v^-)}{\tau})}\right)$$

How to pick positive and negative examples?

- Class labels are usually a good idea
- Example Image Classificaiton

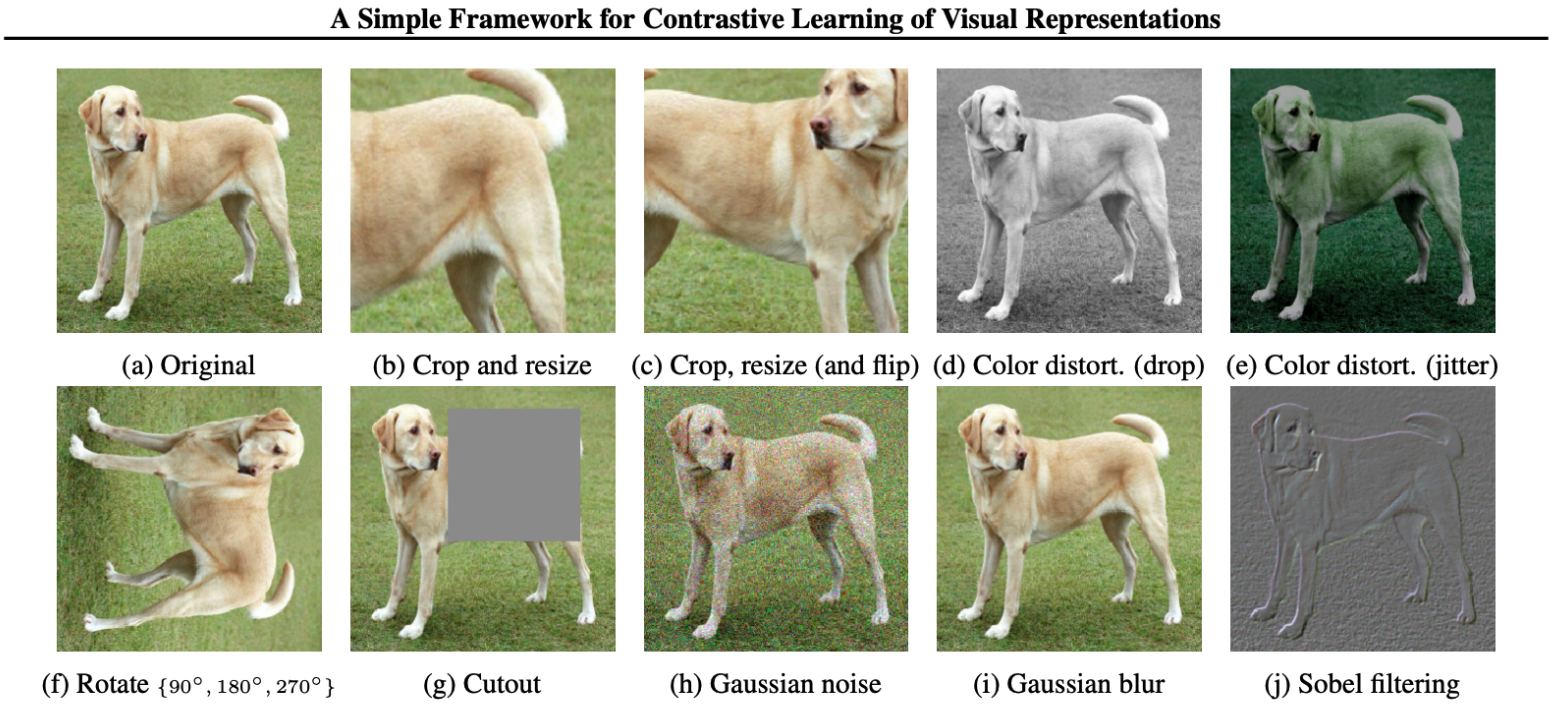


Figure 4. Illustrations of the studied data augmentation operators. Each augmentation can transform data stochastically with some internal parameters (e.g. rotation degree, noise level). Note that we *only* test these operators in ablation, the *augmentation policy used to train our models* only includes *random crop (with flip and resize)*, *color distortion*, and *Gaussian blur*. (Original image cc-by: Von.grzanka)

Figure: Augmenting Images from SimCLR - (Chen, 2020)

SimCLR - Visual

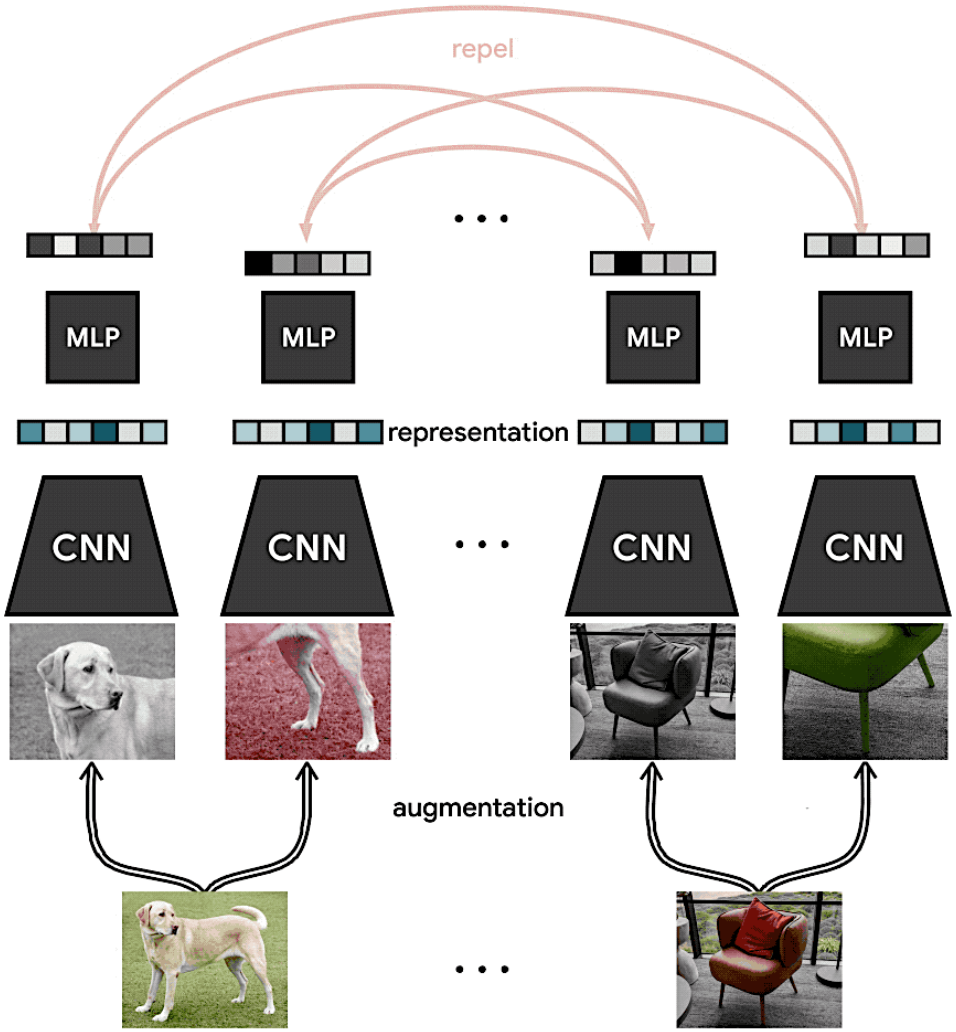


Figure: Augmenting Images from SimCLR - (Chen, 2020)

SimCLR - Algorithm

Algorithm 1 SimCLR's main learning algorithm.

input: batch size N, constant τ, structure of $f, g, \mathcal{T}$.
for sampled minibatch $\{\boldsymbol{x}_k\}_{k=1}^{N}$ **do**
  for all $k \in \{1, \ldots, N\}$ **do**
    draw two augmentation functions $t \sim \mathcal{T}$, $t' \sim \mathcal{T}$
    # the first augmentation
    $\tilde{\boldsymbol{x}}_{2k-1} = t(\boldsymbol{x}_k)$
    $\boldsymbol{h}_{2k-1} = f(\tilde{\boldsymbol{x}}_{2k-1})$ # representation
    $\boldsymbol{z}_{2k-1} = g(\boldsymbol{h}_{2k-1})$ # projection
    # the second augmentation
    $\tilde{\boldsymbol{x}}_{2k} = t'(\boldsymbol{x}_k)$
    $\boldsymbol{h}_{2k} = f(\tilde{\boldsymbol{x}}_{2k})$ # representation
    $\boldsymbol{z}_{2k} = g(\boldsymbol{h}_{2k})$ # projection
  end for
  for all $i \in \{1, \ldots, 2N\}$ and $j \in \{1, \ldots, 2N\}$ **do**
    $s_{i,j} = \boldsymbol{z}_i^\top \boldsymbol{z}_j / (\|\boldsymbol{z}_i\| \|\boldsymbol{z}_j\|)$ # pairwise similarity
  end for
  define $\ell(i, j)$ **as** $\ell(i, j) = -\log \frac{\exp(s_{i,j}/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(s_{i,k}/\tau)}$
  $\mathcal{L} = \frac{1}{2N} \sum_{k=1}^{N} [\ell(2k-1, 2k) + \ell(2k, 2k-1)]$
  update networks f and g to minimize $\mathcal{L}$
end for
return encoder network $f(\cdot)$, and throw away $g(\cdot)$

Figure: Training Algorithm - (Chen, 2020)

Embedding models benefit from contrastive learning

- Goal is to generate a model that can map similar concepts
- Many tasks: classify, cluster, rerank, etc

# of datasets →	Class. 12	Clust. 11	PairClass. 3	Rerank 4	Retr. 15	STS 10	Summ. 1	Avg 56
Unsupervised models								
Glove	57.3	27.7	70.9	43.3	21.6	61.9	28.9	42.0
BERT	61.7	30.1	56.3	43.4	10.6	54.4	29.8	38.3
SimCSE-BERT-unsup	62.5	29.0	70.3	46.5	20.3	74.3	31.2	45.5
E5-PT$_{small}$	67.0	41.7	78.2	53.1	40.8	68.8	**32.7**	54.3
E5-PT$_{base}$	67.9	43.4	79.2	53.5	42.9	69.5	31.1	55.6
E5-PT$_{large}$	69.0	**44.3**	80.3	54.4	44.2	69.9	32.6	56.6
Supervised models								
SimCSE-BERT-sup	67.3	33.4	73.7	47.5	21.8	79.1	23.3	48.7
BERT-FT$_{base}$	68.7	33.9	82.6	50.5	41.5	79.2	29.0	55.2
Contriever	66.7	41.1	82.5	53.1	41.9	76.5	30.4	56.0
GTR$_{large}$	67.1	41.6	85.3	55.4	47.4	78.2	29.5	58.3
Sentence-T5$_{large}$	72.3	41.7	85.0	54.0	36.7	81.8	29.6	57.1
E5$_{small}$	71.7	39.5	85.1	54.5	46.0	80.9	31.4	58.9
E5$_{base}$	72.6	42.1	85.1	55.7	48.7	81.0	31.0	60.4
E5$_{large}$	**73.1**	43.3	**85.9**	**56.5**	**50.0**	**82.1**	31.0	**61.4**
Larger models								
GTR$_{xxl}$	67.4	42.4	86.1	56.7	48.5	78.4	30.6	59.0
Sentence-T5$_{xxl}$	73.4	43.7	85.1	56.4	42.2	82.6	30.1	59.5

Figure: Models and task accuracy - (Wang, 2024)

Training an embedding model - E5

- Typically Three phases
  - Base Model ex BERT
  - Unsupervised/SemiSupervised Pre-training
  - Supervised training

data source	type of text pairs	random example	# of pairs
Wikipedia	(entity+section title, passage)	**q**: Lexden History **p**: The site on which Lexden now stands was crossed by the fortifications of iron age Colchester...	24M
Reddit	(post, upvoted comment)	**q**: What makes a client good quality to you? I'm putting together my ideal client ... **p**: Respectful of schedules. And pays on time....	60M
Common Crawl	(title, passage)	**q**: Central Intake Unit | Broome County **p**: Caseworkers from Central Intake assess the household and risk of placement. If eligible...	69M
Stackexchange	(title, answer) (title+description, answer)	**q**: Will killing Python made problems for Apache **p**: Python and Apache aren't related, unless your app is making use of Python. ...	19M
S2ORC	(title, abstract) (title, citation title) (abstract, citation abstract)	**q**: Constructive Dual DP for Reservoir Optimization **p**: Dynamic programming (DP) is a well established technique for optimization of reservoir manage...	90M
News	(title, passage) (highlight, passage)	**q**: LG Display reports Q1 operating loss as... **p**: April 25 (Reuters) - South Korea's LG Display Co Ltd reported its first quarterly operating loss...	3M
Others	misc.	misc.	6M
All above	-	-	$\sim$ 270M

Figure: Weakly Supervised Pre-training - (Wang, 2024)

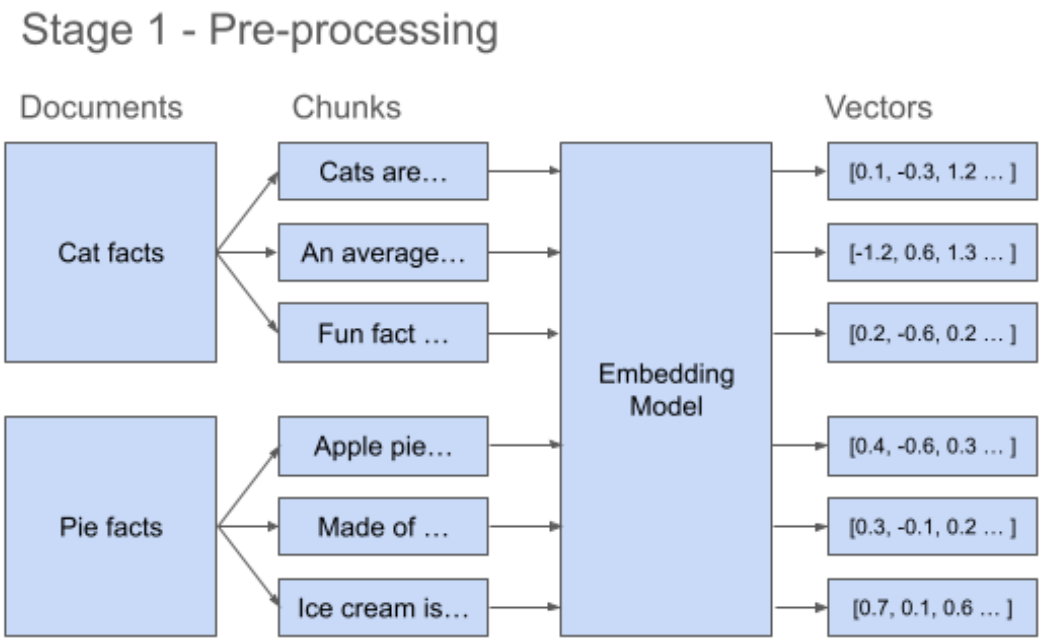
Intro to RAG - Retrieval

- RAG is used as a way to provide a LLM with more relevant context to complete a task
- RAG Architecture - use an embedding model to retrieve relevant information
- Usually this means the LLM objective shifts from generation to summarization
- Commonly extensions are to use Keyword searches or even graph structures with available

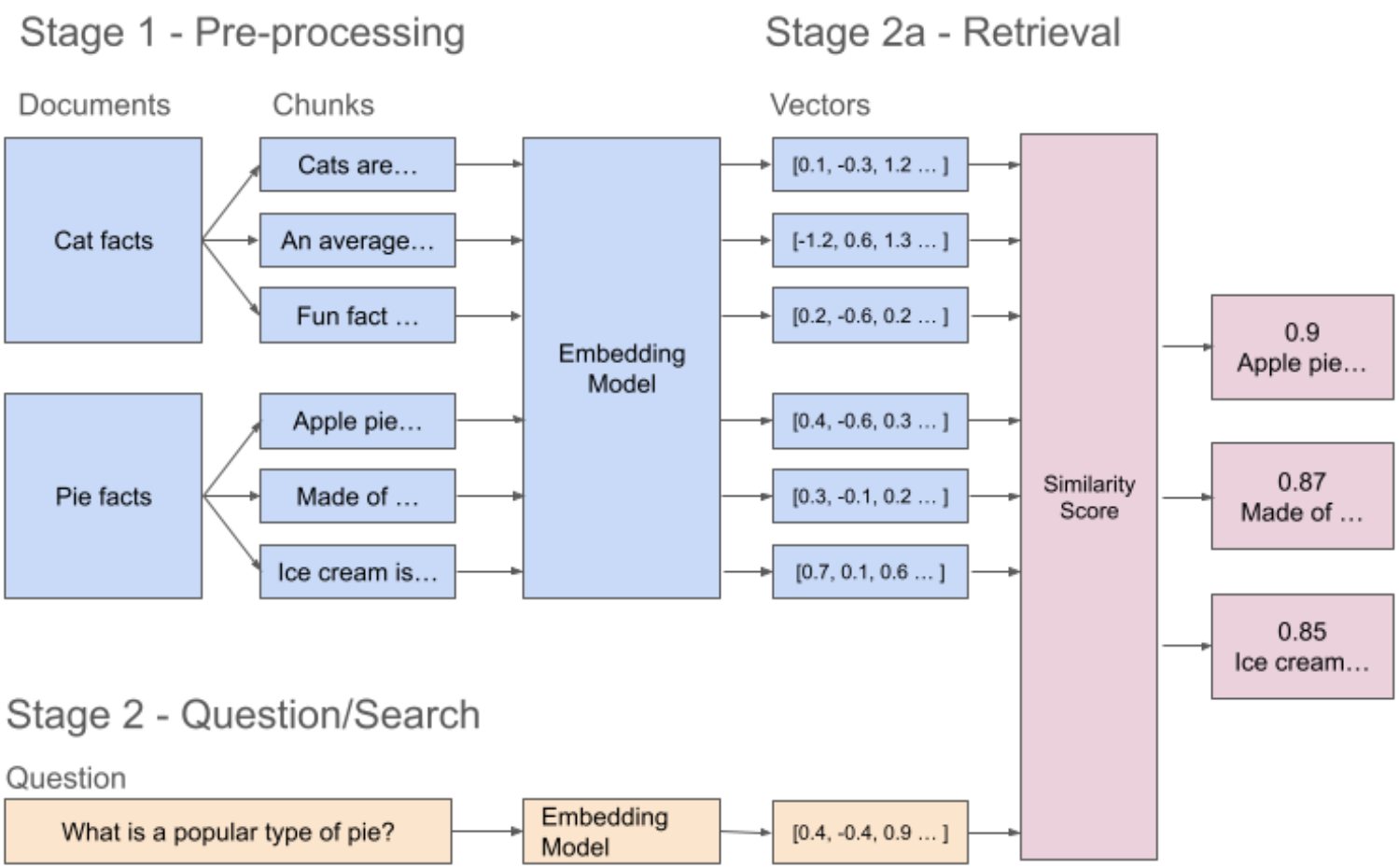
RAG Architecture

- RAG is used as a way to provide a LLM with more relevant context to complete a task
- Contains three main parts:
  - Preprocess information into searchable form
  - Retrieve relevant chunks of information
  - Generate using LLM based on prompt and chunks

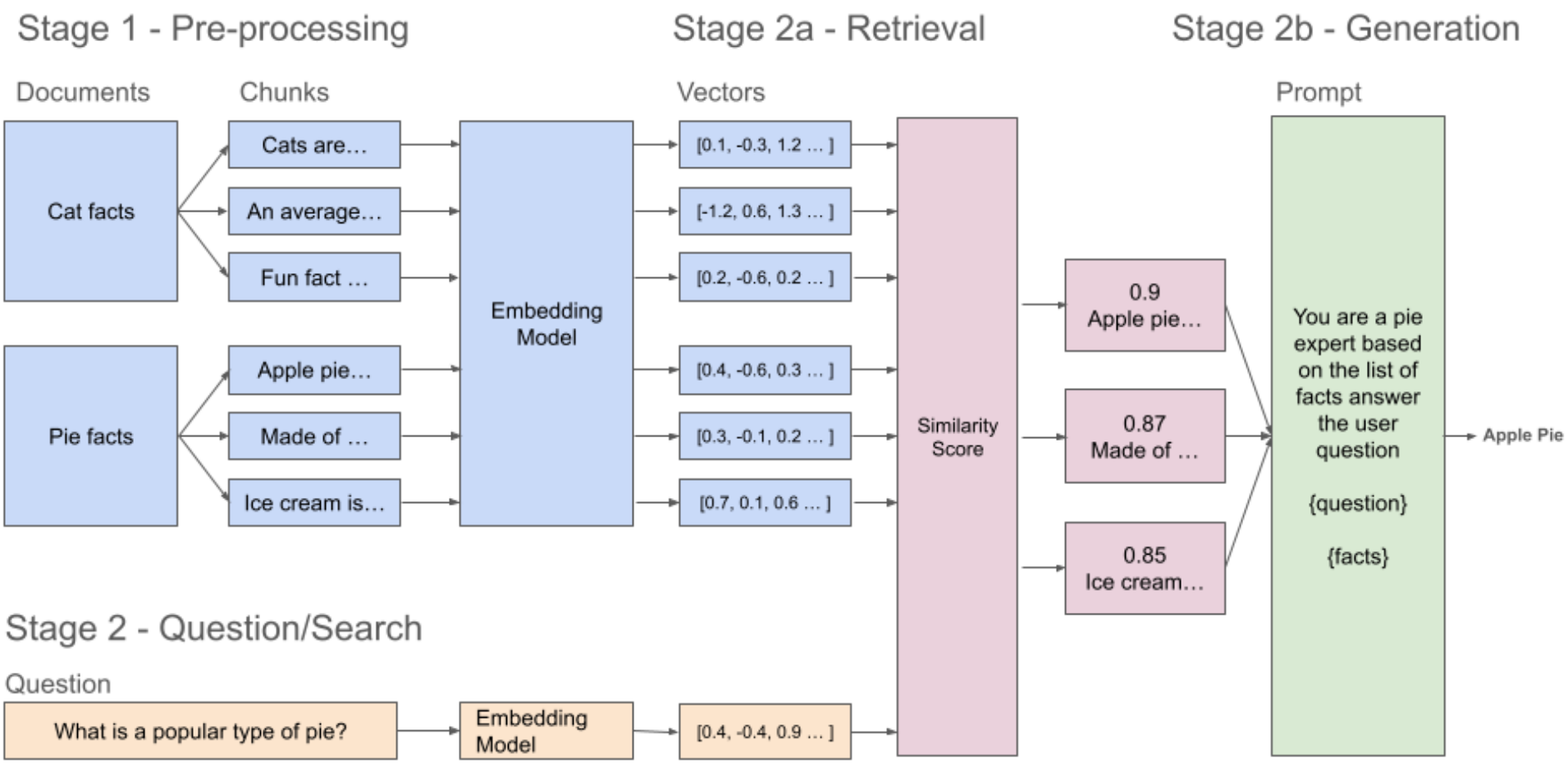
RAG Architecture



RAG Architecture



RAG Architecture



Summary

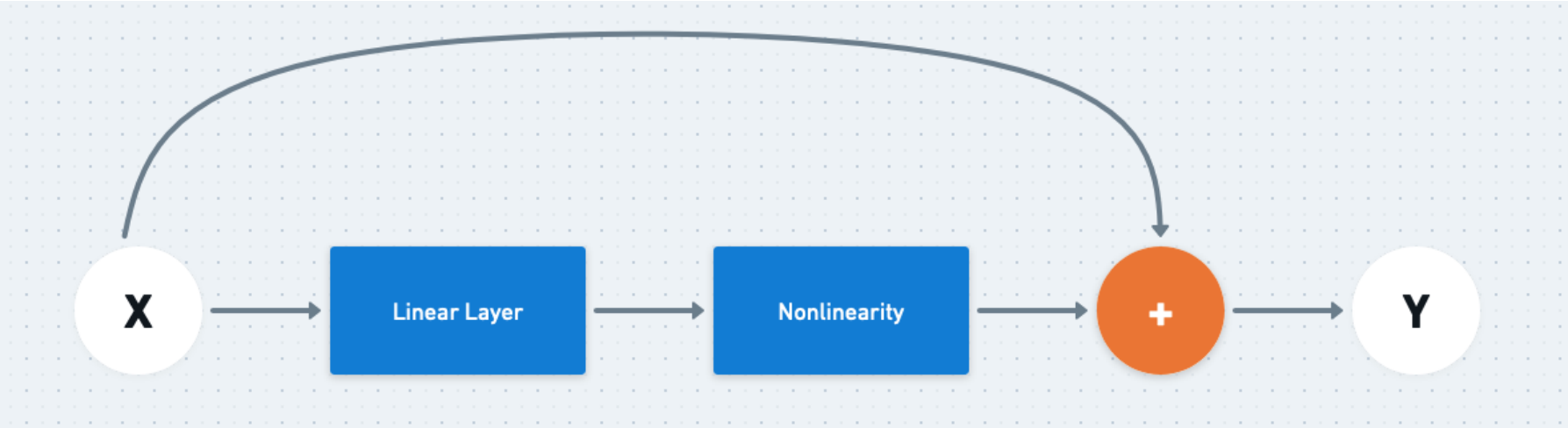
- The first modelling step with any data should be to convert it to a numerical representation.
- We can learn embeddings from unlabelled data.
- We can easily combine different kinds of embeddings to improve how we represent data.
- We have learned a number of different document, and token embedding algorithms.
- Human Language is hard!

Part 2

- Neural Network Building Blocks
  - Residual Layers
  - Recurrent Layers
  - **Attention**
- Neural Networks
  - Recurrent
  - **Transformer**
- Transformer
  - Encoder
  - Decoder
  - Positional Encoding
  - GPT
  - Attention improvements
  - RAG

Building Blocks of Neural Networks

If we begin stacking large number of layers together, the signal may get squashed to zero, or blow up to infinity. Similar problem often happens during the gradient computation back through the graph. To reduce the effect of those problems we often propagate the signal to layers further downstream, in what are called **residual connections**



$$y = f(x;\theta) = \phi(Wx + b) + x$$

Building Blocks of Neural Networks

When it comes to modelling sequential data (e.g. text, time series), it is often useful to make the model stateful in order for it to help "carry" the information through the graph. To do that we simply add a state at timepoint t: s_t, and computing the output and the new state using some function:

$$(y, s_{t+1}) = f(x, s_t)$$

This is then called a **recurrent layer**.

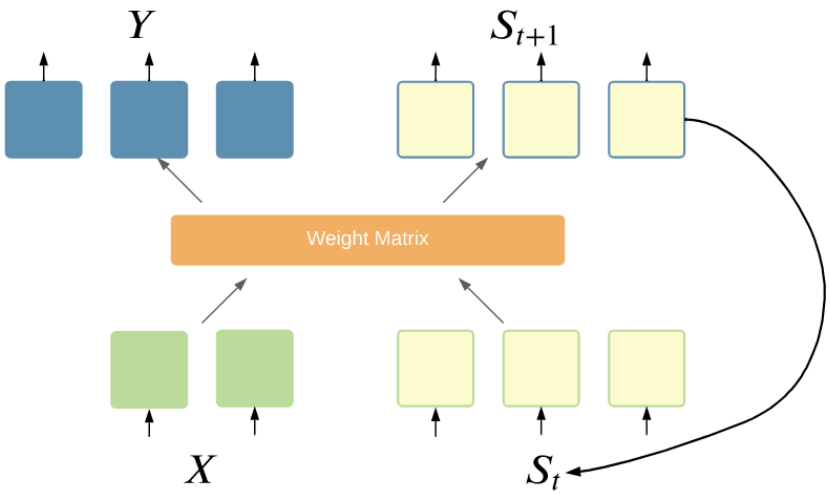


Figure 16.8: Recurrent layer.

RNN

If we use recurrent layers in our neural network, the outcome is what we typically call a **Recurrent Neural Network**, (of which there are many variants). In the simplest possible option the function $f(x, h)$ is a simple FFNN. When training RNNs each item in a sequence is used as input, however during inference each item in the sequence will depend on previous predictions.

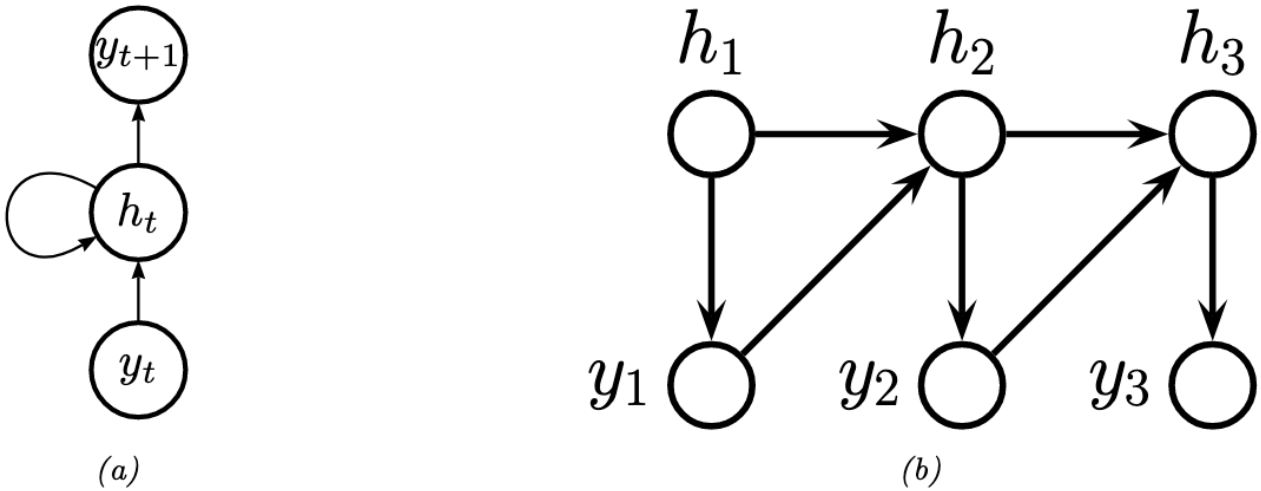


Figure 16.12: Illustration of a recurrent neural network (RNN). (a) With self-loop. (b) Unrolled in time.

Attention is all you need

What if instead of getting just the previous hidden state we were able to take a look at a lot of the previous inputs at once? We could combine all the previous hidden states. But can we do better? We can score each of the hidden states by how well it is associated with the state we will be predicting. At a high level the attention mechanism consists of 3 simple steps:

1. Generate a score for each of the hidden states
2. Apply the softmax function to the scores
3. Multiply each of the hidden states by the output of the softmax and add them together.

Attention is all you need

- We can create hidden states by learning different representations
- Create the Query, Key and Value Matricies
- Comes from an information retrieval background

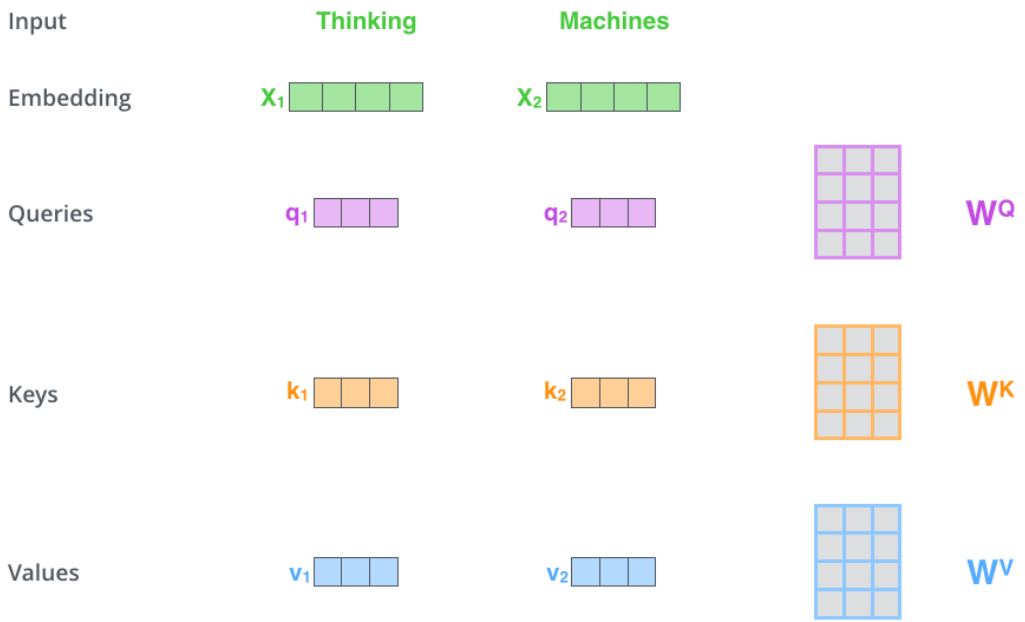


Figure: Illustrated Transformer - (Alammar, 2018)

Attention is all you need

Now the hard problem remains: how do we score each of the hidden states? We will begin by creating 3 separate embeddings from each of our inputs, by simply multipling them by (learned) matrices:

$$q = W^Q x$$
$$k = W^K x$$
$$v = W^V x$$

We then define the **Attention Layer** as:

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

Where α is the scoring function.

(Dot product) Attention is all you need

$$Attn(q, k, v) = \sum_{i=1}^{m} \alpha_i(q, k_i) v_i$$

The most common choice of the attention function is called the **dot product attention**. We obtain the scores by a normalized dot product of the k and q vectors.

$$b(q, k) = \frac{q^T k}{\sqrt{d}}$$

where d is a normalizing constant, usually the dimensionality of the vectors. We then set our attention weights α_i to be the softmax of all the scores:

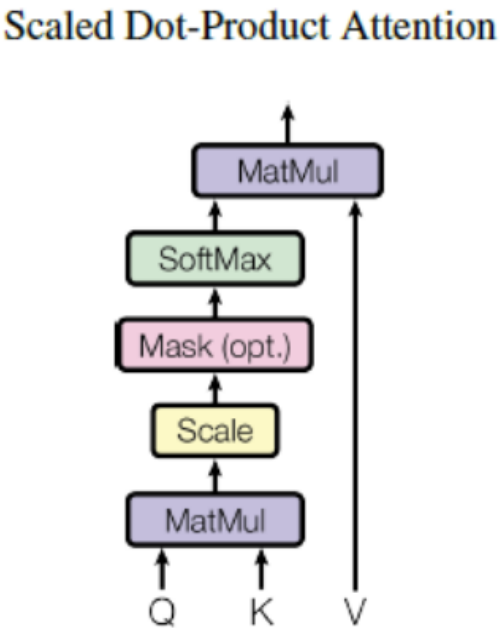
$$\alpha_i(q, k_i) = \frac{exp(b(q, k_i))}{\sum_{j=1}^{m} exp(b(q, k_j))}$$

(Dot product) Attention is all you need

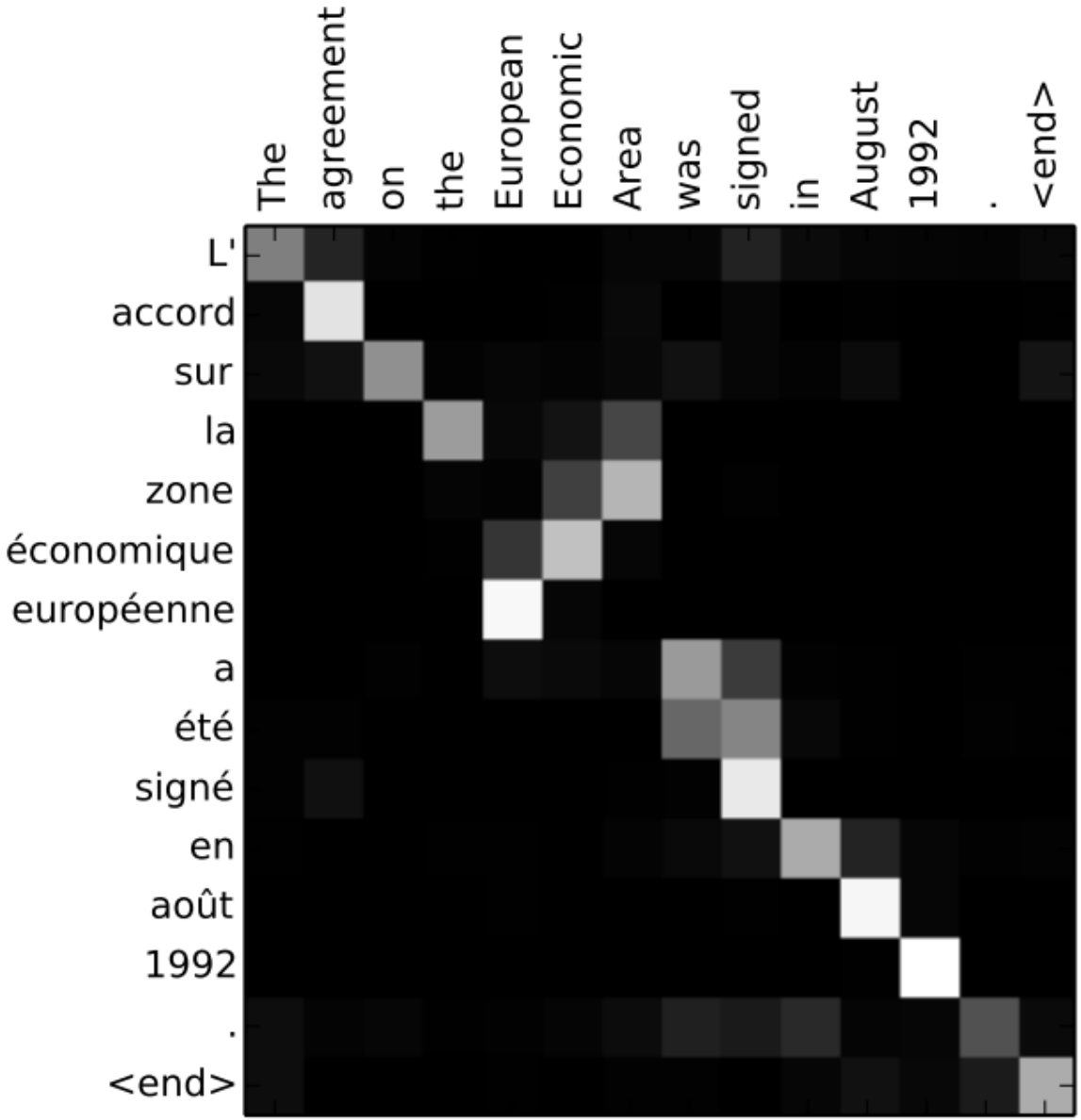
The entire process then reduces to:

$$Y = Attn(Q, K, V) = \sigma(\frac{QK^T}{\sqrt{d}})V$$

$$= \sigma(\frac{W^Q X (W^K X)^T}{\sqrt{d}})W^V X$$



Attention Visualization



Multi Head Attention and Self Attention

In practice it is advantageous to have multiple "attention heads" each with a different set of W^Q, W^K, W^V matrices.

- Why do you think that is?
- Do we really need all of them?

We then simply concatenate the outputs of all of the attention heads together and multiplied by one final matrix W^O that is learned as well, this is called **Multi Head Attention**.

$$\begin{aligned} o &= MHA(Q, K, V) = Concat(h_1, \ldots, h_h)W^O \\ &= Concat(Attn(Q_1, K_1, V_1), \ldots, Attn(Q_h, K_h, V_h))W^O \end{aligned}$$

Additionally, we can stack several identical Attention / MHA blocks on top each other.

MHA Illustration

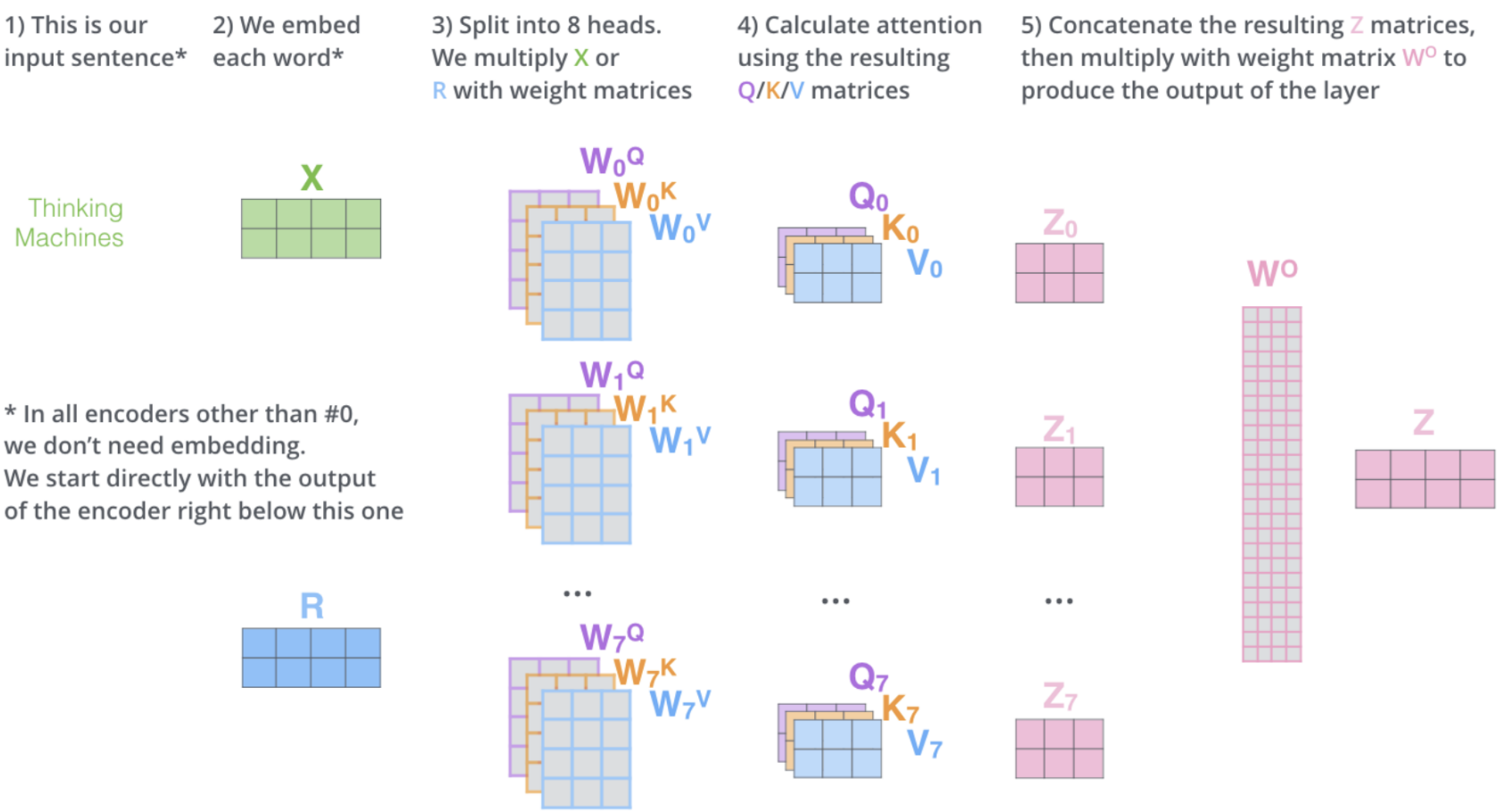


Figure: Illustrated Transformer - (Alammar, 2018)

Transformer

First proposed in a 2017 paper "Attention is all you need", the **Transformer** architecture consists of two stacks (called **Encoder** and **Decoder**) of blocks:

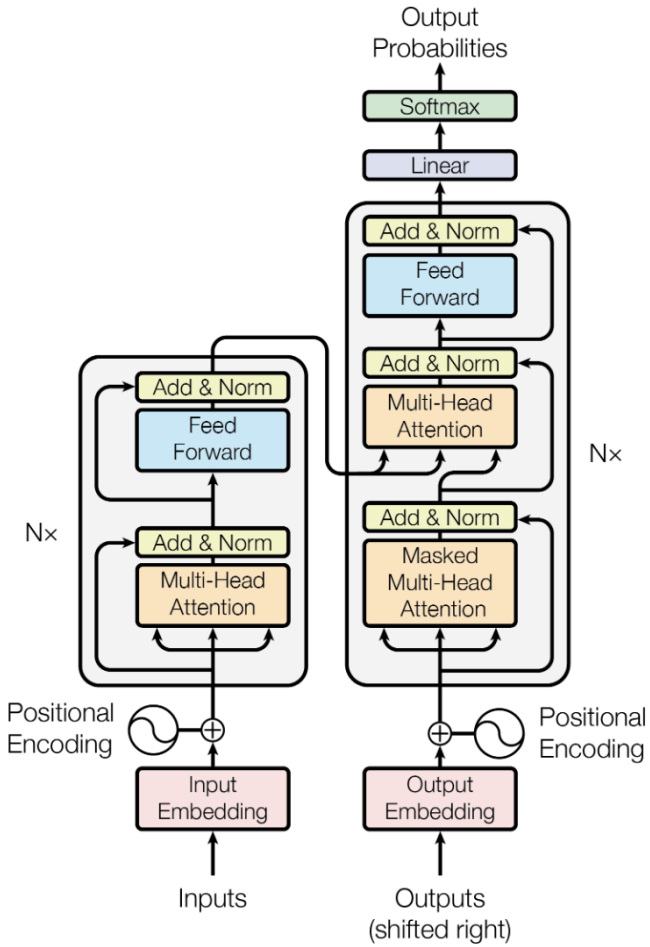
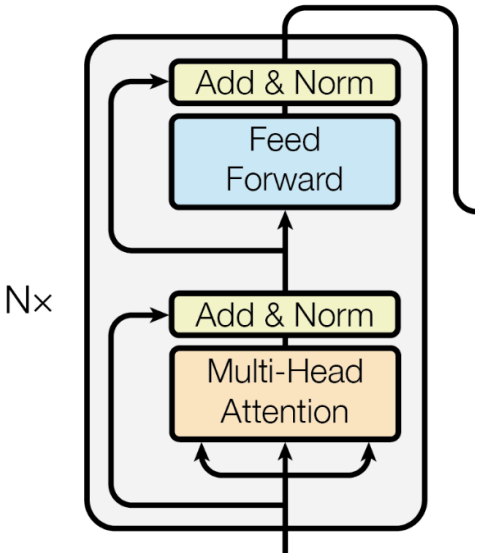


Figure 1: The Transformer - model architecture.

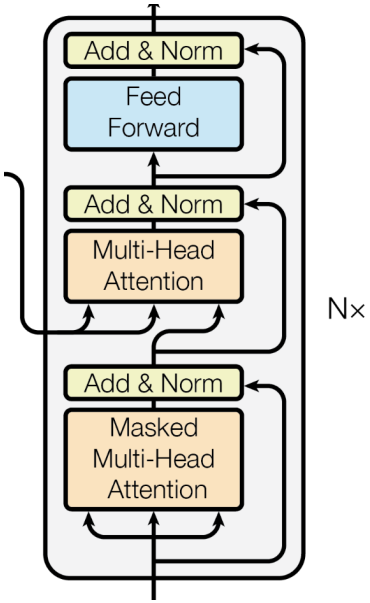
Transformer

- The **Encoder** consists of a stack of 6 blocks. Each block is further split into two distinct sub-blocks.
- The first is a Multi Head Self Attention mechanism, and the second is a simple FFNN. Both of the sub-blocks have a residual connection around them, followed by normalization.



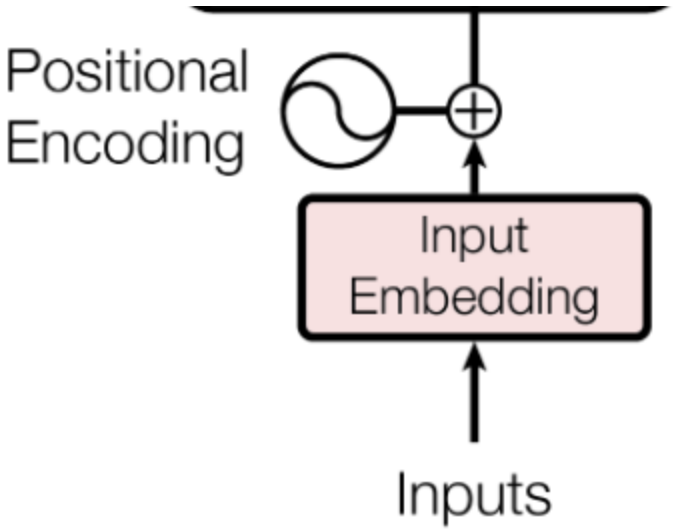
Transformer

- Similarily, the **Decoder** is also a stack of 6 blocks.
- However in addition to the two sub-blocks of the encoder, it features a 3rd sub-block.
- This 3rd sub-block performs multi-head attention over the output of the encoder. This "encoder-decoder attention" layer uses Q from the previous decoder layer, and K, V from the output of the encoder.

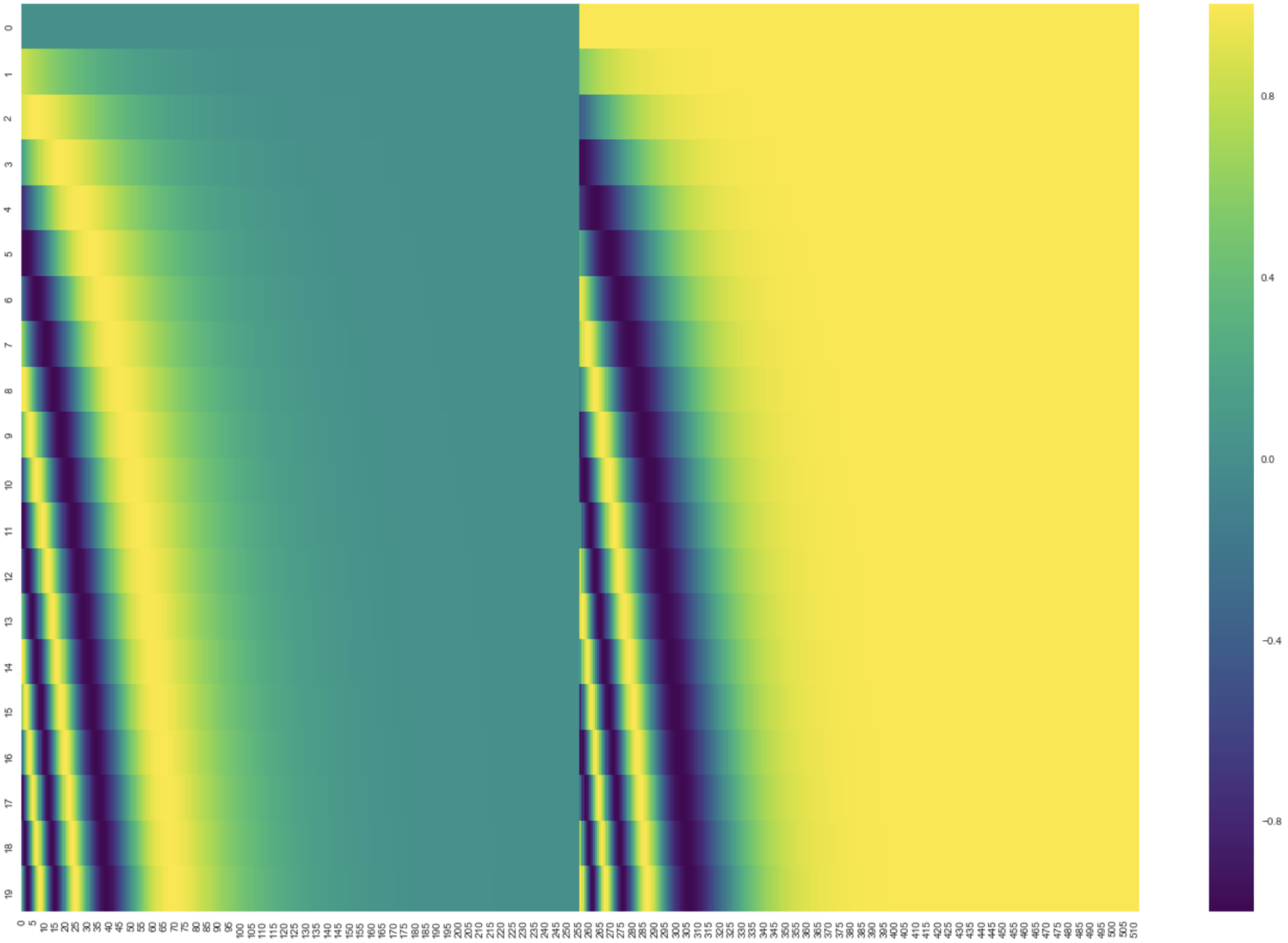


Transformer

- What about inputs?
- The input embedding is a learneable "static" **token** embedding similar to the Word2Vec model we have seen in the lecture 9.
- What is "Positional Encoding?"
- It's either a learneable (representing position in a sequence) embedding, or a predefined embedding.

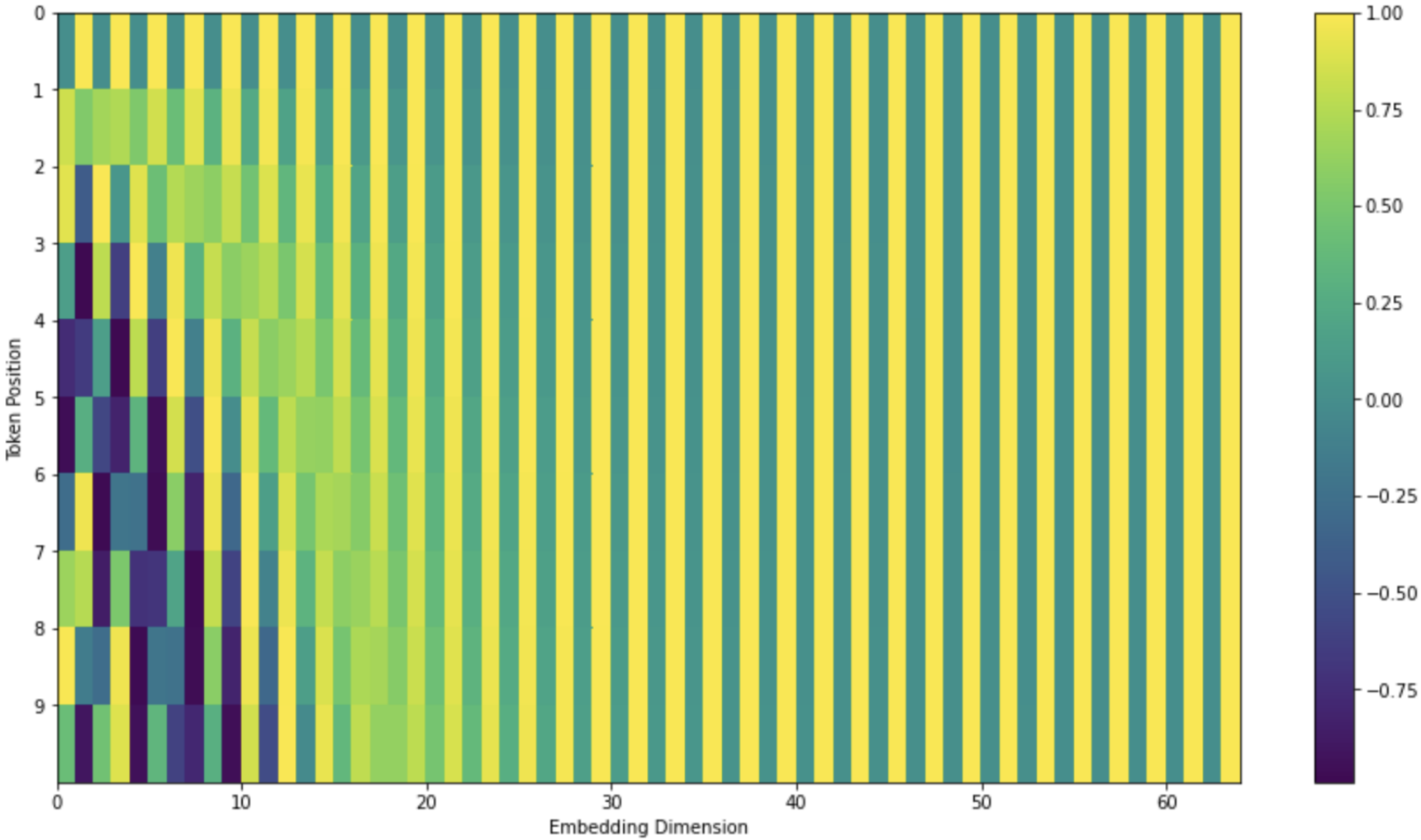


Positional Encoding



A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in

Positional Encoding



How to train a Transformer

- The original Transformer model was trained on an English $\leftrightarrow$ German translations, where at each step the final decoder state was fed into a simple Linear Layer followed by a softmax to produce probabilities over next tokens.
- Currently there are a large number of pre-training tasks (similar in idea to W2V). One of the most common ones is **Masked Language Modelling**, where we randomly replace 15% of tokens with "[MASK]", and the goal of the model is to predict back the original token. BERT (Devlin, 2018) used this approach
- The other approach is to predict the next token based on past tokens only. GPT uses this architecture (Radford, 2018)

GPT-1 Exerpt (Paraphrase)

Given an unsupervised corpus of tokens $\mathcal{U} = \{u_1, \ldots, u_n\}$, maximize the following likelihood:

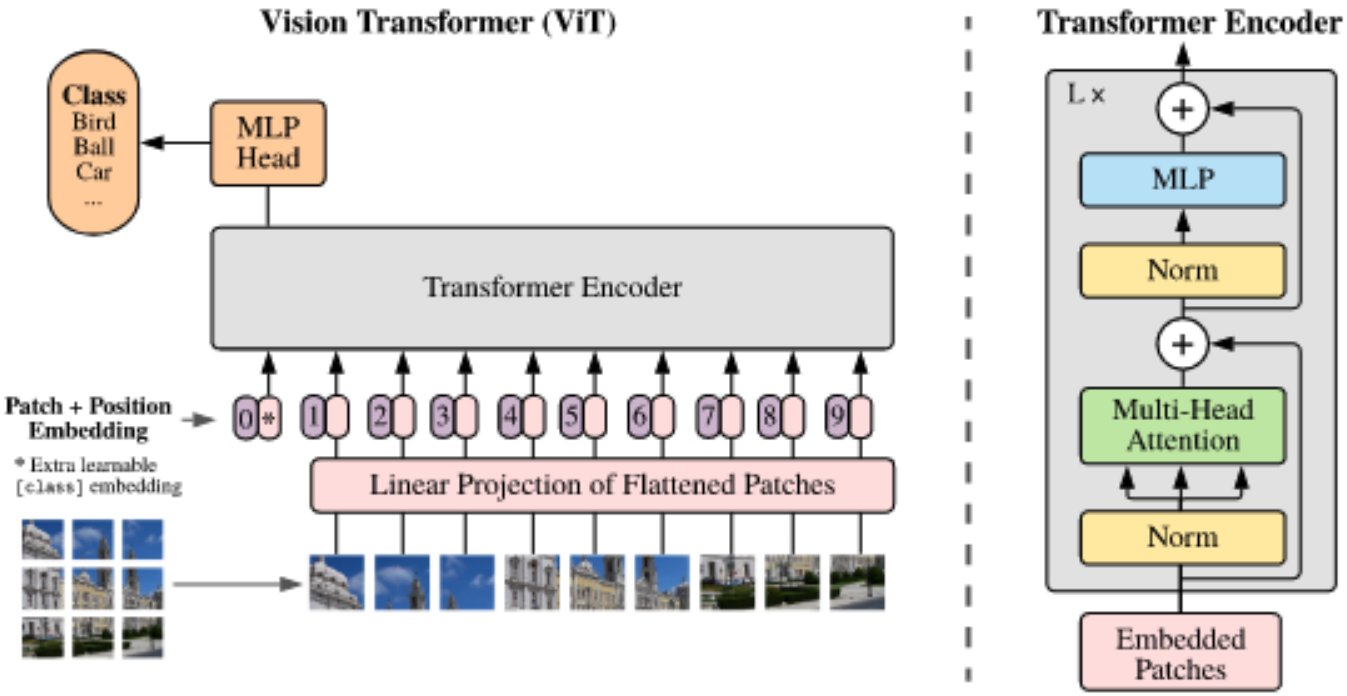
$$L_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta) \tag{1}$$

where k is the size of the context window, and the conditional probability P is modeled using a neural network with parameters Θ. Use multi-headed self-attention operation over the input context tokens followed by position-wise feedforward layers to produce an output distribution over target tokens:

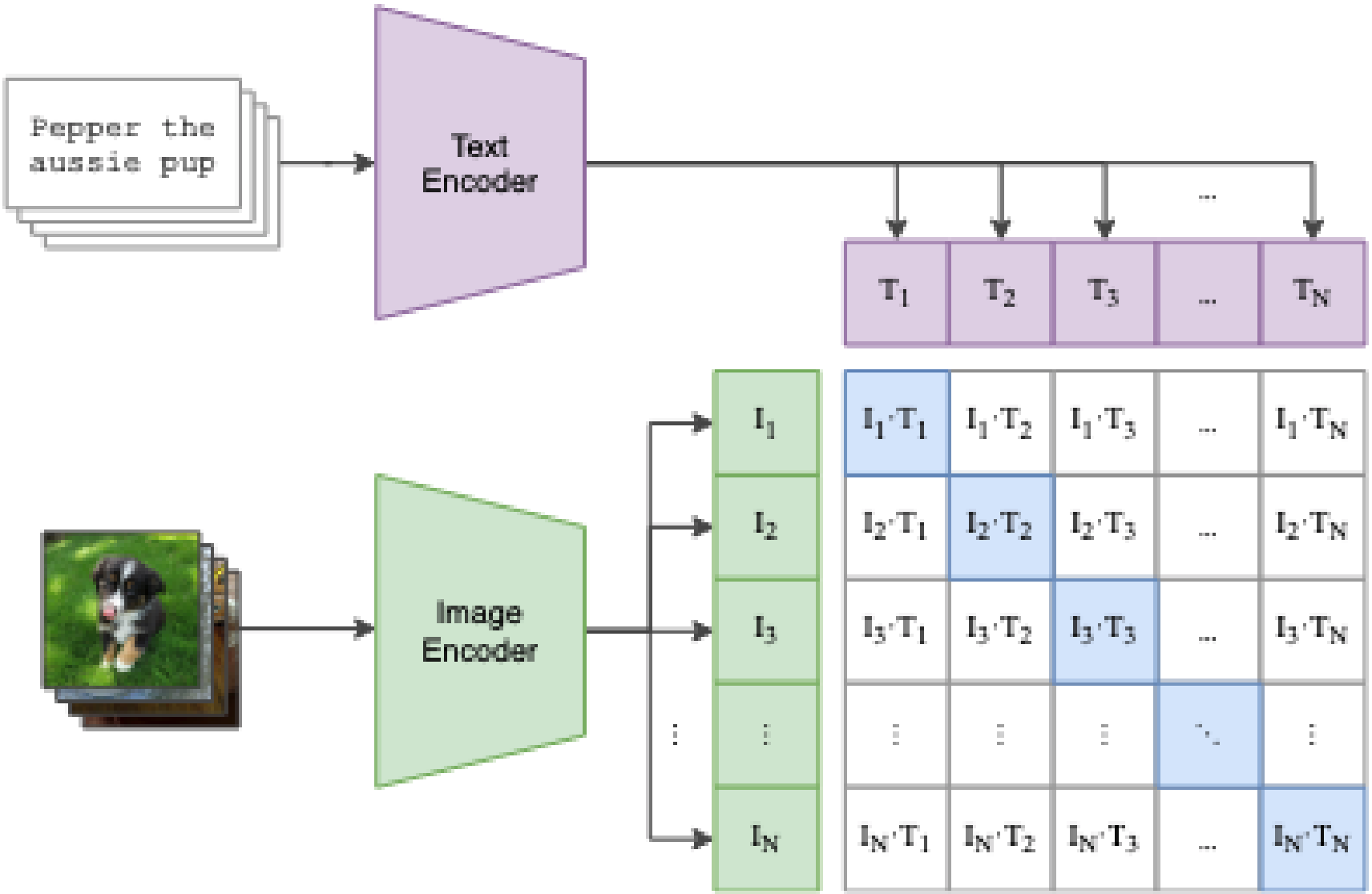
$$h_0 = UW_e + W_p$$
$$h_l = \text{transformer\_block}(h_{l-1}) \quad \forall l \in [1, n]$$
$$P(u) = \text{softmax}(h_n W_e^T)$$

where $U = (u_{-k}, \ldots, u_{-1})$ is the context vector of tokens, n is the number of layers, W_e is the token embedding matrix, and W_p is the position embedding matrix.

Vision Transformers



CLIP


Pepper the aussie pup
Text Encoder
T1
T2
T3
...
TN
Image Encoder
I1
I2
I3
IN
I1'T1
I1'T2
I1'T3
I1'TN
I2'T1
I2'T2
I2'T3
I2'TN
I3'T1
I3'T2
I3'T3
I3'TN
IN'T1
IN'T2
IN'T3
IN'TN

Are RNNs Dead?

v6 7B Demo | v7 3B Demo | WebGPU Demo

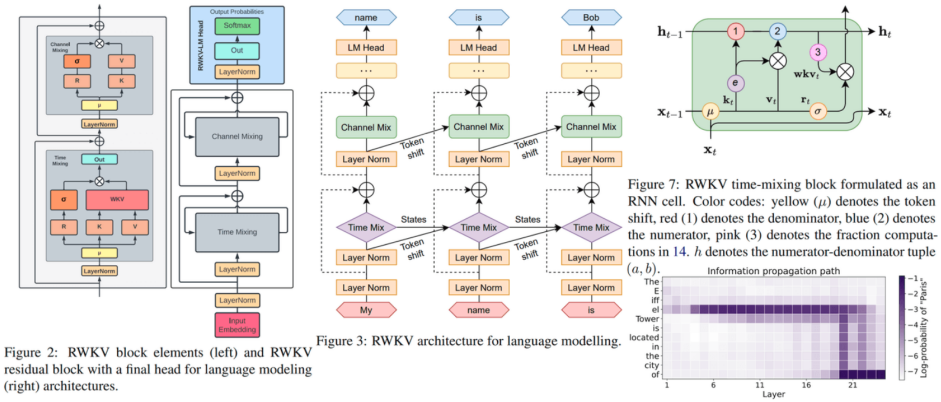


Figure 2: RWKV block elements (left) and RWKV residual block with a final head for language modeling (right) architectures.

Figure 3: RWKV architecture for language modelling.

Figure 7: RWKV time-mixing block formulated as an RNN cell. Color codes: yellow (μ) denotes the token shift, red (1) denotes the denominator, blue (2) denotes the numerator, pink (3) denotes the fraction computations in 14. h denotes the numerator-denominator tuple (a, b).

Figure: RWKV - Attention free LLMs (Peng, 2021)

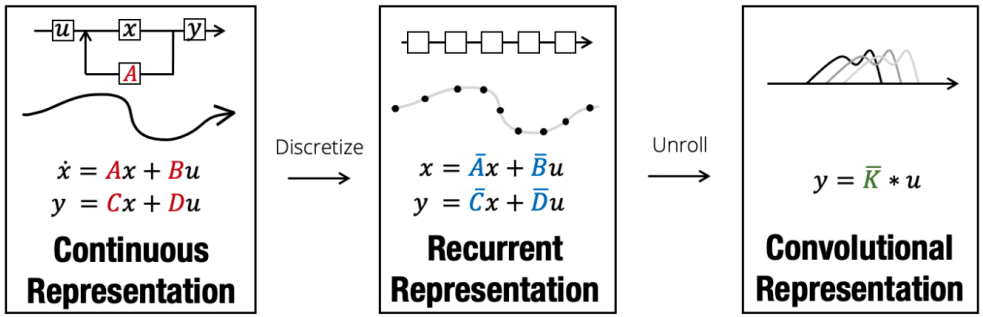


Figure: S4 State Space Model - (Gu, 2021)

Modern Attention

- Attention is expensive because it is quadratic in length
- Linear attention - ex Sliding Windows

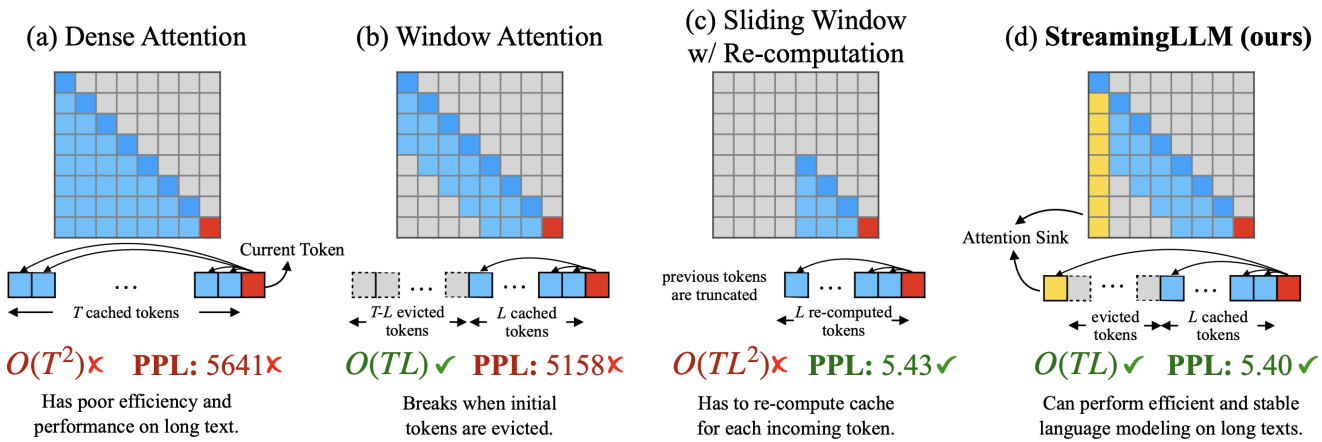


Figure: Attention Sink Example - (Xiao, 2023)

- Grouped Query Attention (Ainslie, 2023)
- Multi Latent Attention (Deepseek, 2024)
- Flash Attention - GPU optimization (Dao, 2022)
- Many LLMs use GQA (but architecture design details are sparse)

Other questions to think about

- Different types of attention, tradeoffs
- Embedding vector size, precision and dimension
- Limitations of RAG
- Different forms of LLM training
- Model training vs inference differences
- Intercection of hardware and software