

Lecture 7: word embeddings, attention, transformers

Jikael Gagnon (Based on slides by Sheheryar Parvaz and Shidan Javaheri)

McGill Artificial Intelligence Society

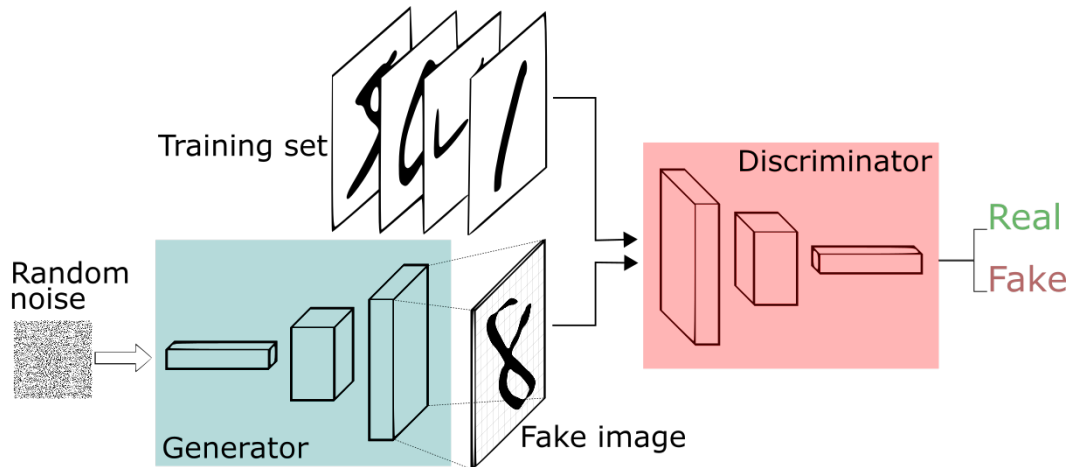
Winter 2026



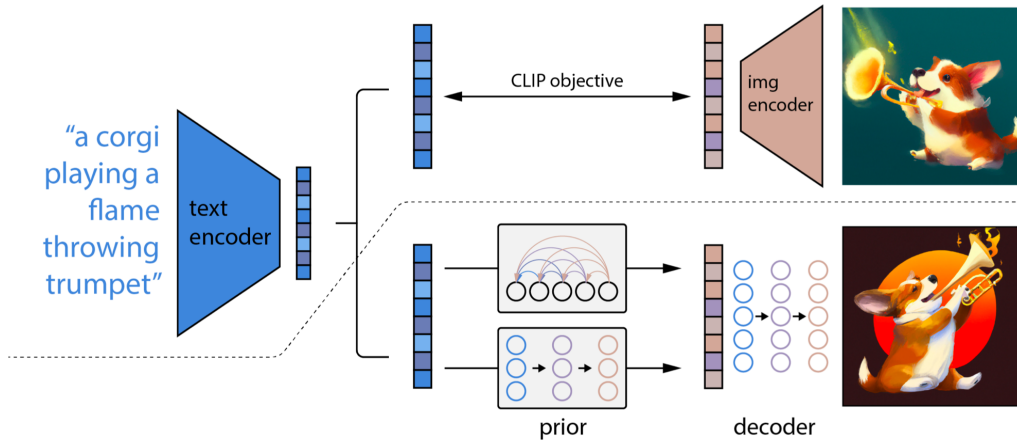
Announcements

- A3 released today!
- Deliverable 4 released today!
- Next week's lecture will be a special topic chosen by Ritchie. He will (likely) cover GANs and diffusion models

Generative adversarial networks (GANs)



Diffusion



word embeddings

Recall how we've encoded words so far: **one-hot encoding**.

$$\text{student} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

Recall how we've encoded words so far: **one-hot encoding**.

$$\text{student} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

These are a good starting point but are often not a great choice...

In additive attention, we saw the idea of using a function to score the similarity between two vectors:

$$\text{score}(\vec{x}, \vec{y})$$

In additive attention, we saw the idea of using a function to score the similarity between two vectors:

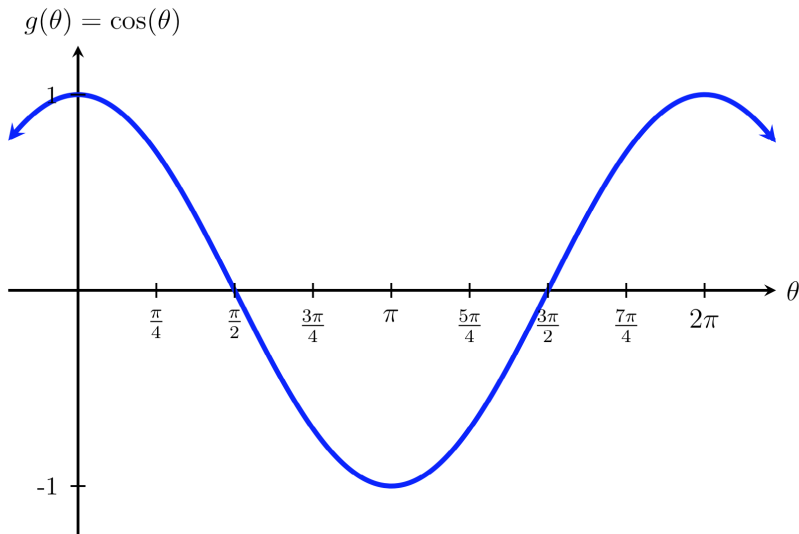
$$\text{score}(\vec{x}, \vec{y})$$

A common choice for this is the *cosine similarity*:

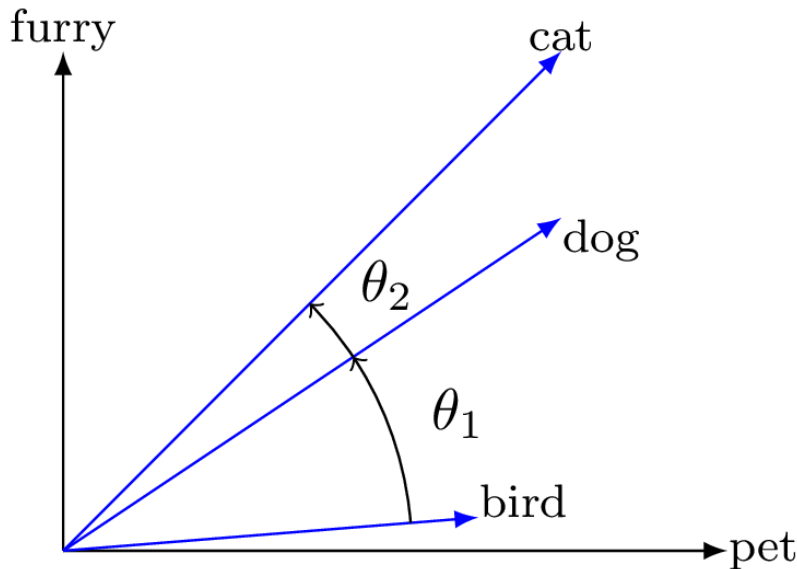
$$\text{cosine-sim}(\vec{x}, \vec{y}) = \cos(\theta) = \frac{\vec{x}^T \cdot \vec{y}}{||\vec{x}|| ||\vec{y}||}$$

where θ is the angle between $\vec{x}$ and $\vec{y}$ and ranges from 0 to π .

This score tells us the cosine of the angle between two vectors. Note that $\cos$ is monotonically decreasing in the range $[0, \pi]$:



Since $\cos$ is monotonically decreasing in the range $[0, \pi]$, then vectors with a smaller angle θ between them will get a *larger* score:



Another common way of scoring similarity is via dot-product similarity:

$$\text{dp-sim}(\vec{x}, \vec{y}) = \vec{x}^T \cdot \vec{y}$$

(This is very similar to before except we aren't normalizing by the product of the lengths)

This type of comparison is meaningless for one-hot encodings since different words will always get a similarity of 0.

This type of comparison is meaningless for one-hot encodings since different words will always get a similarity of 0.

Idea: Instead, let's try to *learn* representations of words that *embed* meaning. We will call these representations *word embeddings*.

$$\text{student} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} -0.64 \\ 1.3 \\ \vdots \\ 0.74 \end{bmatrix}$$

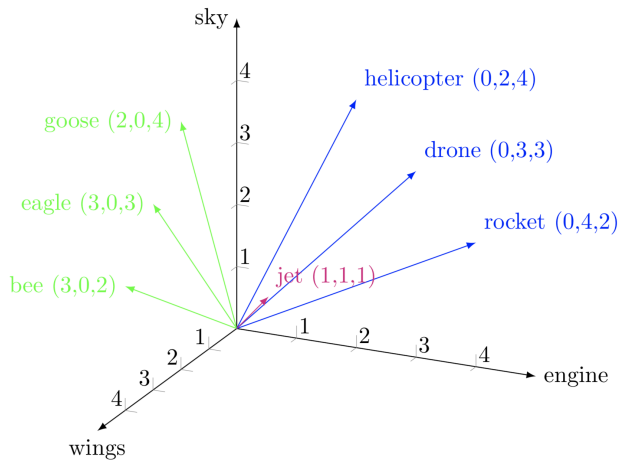
This type of comparison is meaningless for one-hot encodings since different words will always get a similarity of 0.

Idea: Instead, let's try to *learn* representations of words that *embed* meaning. We will call these representations *word embeddings*.

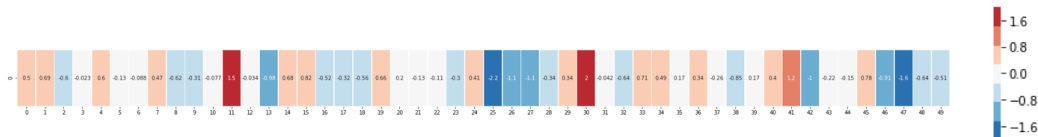
$$\text{student} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} -0.64 \\ 1.3 \\ \vdots \\ 0.74 \end{bmatrix}$$

Popular options include **word2vec**, **GloVe**, **ELMo**, and many others. We will cover word2vec and ELMo. Let's see some examples first.

We can think of each dimension of a word embedding as encoding information about some concept:



Let's take a look at an example of some embeddings generated using the GloVe model. The model takes words as input and embeds them as 50-dimensional vectors:



Negative values are blue and positive values are red.

We can compare these to embeddings of related words. Analyzing this column-wise we can find dimensions that encode similar information (same colour = same info):

“king”

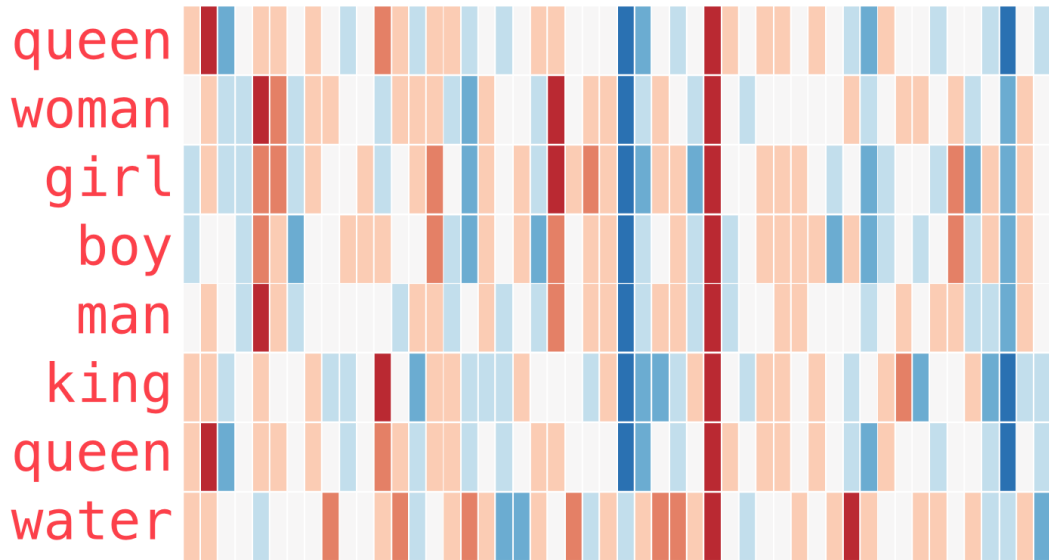


“Man”



“Woman”





If these embeddings are really good, we can even add and subtract embeddings of words to get (close) to new embeddings:

king - man + woman $\approx$ queen



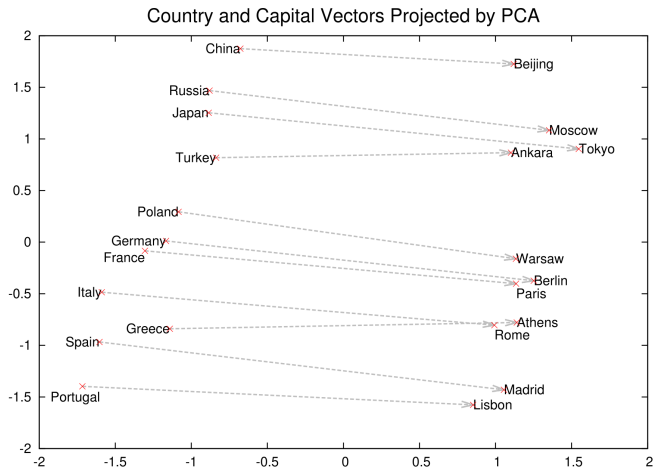


Figure 2: Two-dimensional PCA projection of the 1000-dimensional Skip-gram vectors of countries and their capital cities. The figure illustrates ability of the model to automatically organize concepts and learn implicitly the relationships between them, as during the training we did not provide any supervised information about what a capital city means.

```
>> import gensim.downloader
>>
>> model = gensim.downloader.load('word2vec-google-news-300')
>> model.most_similar(positive=['larger', 'small'], negative=['large'], topn=5)

[('smaller', 0.7564553618431091),
 ('bigger', 0.6463430523872375),
 ('tiny', 0.5519010424613953),
 ('tinier', 0.5254257917404175),
 ('littler', 0.5094303488731384)]
```

```
>> model.most_similar(positive=['bigger', 'small'], negative=['big'], topn=5)

[('larger', 0.7402471899986267),
 ('smaller', 0.7329993844032288),
 ('tiny', 0.5698219537734985),
 ('tinier', 0.5439693927764893),
 ('large', 0.5191665291786194)]
```

```
>> model.most_similar(positive=['Canada'], topn=5)
```

```
[('Canadian', 0.7513012290000916),  
 ('Ontario', 0.6928848028182983),  
 ('Nova_Scotia', 0.6792765259742737),  
 ('Manitoba', 0.67861008644104),  
 ('Alberta', 0.6736730337142944)]
```

word2vec

We will explore how **word2vec** works. Note that there are 2 models, each with 2 variants:

models	variants
continuous bag of words	hierarchical softmax negative sampling
skip-gram	hierarchical softmax negative sampling

word2vec

We will explore how **word2vec** works. Note that there are 2 models, each with 2 variants:

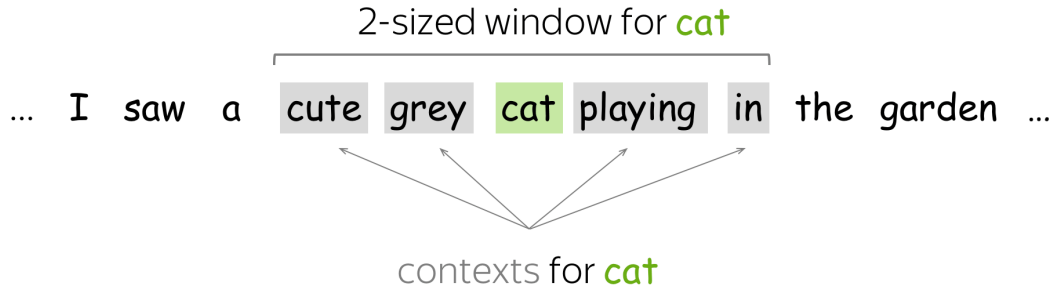
models	variants
continuous bag of words	hierarchical softmax negative sampling
skip-gram	hierarchical softmax negative sampling

We will only cover **skip-gram** with **negative sampling** as it's the most interesting one. The others are quite similar.

Idea: We will train the model to predict likely surrounding (neighbouring) words.

Idea: We will train the model to predict likely surrounding (neighbouring) words.

How? We will create a dataset by looking at each word $\vec{w}_t$ and surrounding words within a certain distance m . We can think of this as a *window* centered at $\vec{w}_t$ and spanning the range $\vec{w}_{t-m}, \vec{w}_{t-(m-1)}, \dots, \vec{w}_{t+(m-1)}, \vec{w}_{t+m}$.



Note: When a word is in the center of the window (input) we will call it a *center word*, and when it's one of the neighbours inside the window we will call it a *context word*.

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

input word	target word
not	thou
not	shalt
not	make
not	a

We will slide this window across each sentence in our dataset to create a dataset of input-outputs pairs.

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness

our dataset

Once we have our dataset, we can train our model:

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness

not →



our model

Suppose we have a vocabulary of V different words:

- Input: $\vec{w} \in \{0, 1\}^V$ (one-hot encoding of center word)
- Output: $\vec{y} \in [0, 1]^V$ (vector of probabilities of each possible context word)

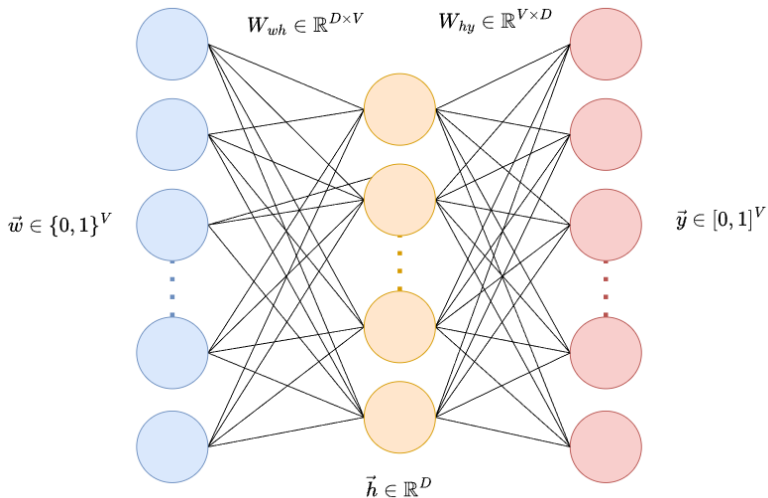
our model

Suppose we have a vocabulary of V different words:

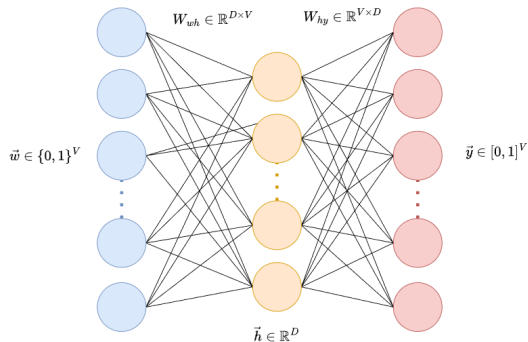
- Input: $\vec{w} \in \{0, 1\}^V$ (one-hot encoding of center word)
- Output: $\vec{y} \in [0, 1]^V$ (vector of probabilities of each possible context word)

This is just multi-class classification! As usual, our loss function will just be cross-entropy loss.

For a vocab of size V , we have a V -dimensional one-hot encoding as input and a V -dimensional probability vector as output. We have a single D -dimensional hidden layer, where D is the size of our embeddings:



Important: There is no activation function in the hidden layer!!!



- $\vec{h} = W_{wh} \vec{w} \in \mathbb{R}^D$
- $\vec{u} = W_{hy} \vec{h} \in \mathbb{R}^V$
- $\vec{y} = \text{softmax}(\vec{u}) \in [0, 1]^V$

This model just predicts probabilities of neighbouring words. How do we get the word embeddings?

This model just predicts probabilities of neighbouring words. How do we get the word embeddings?

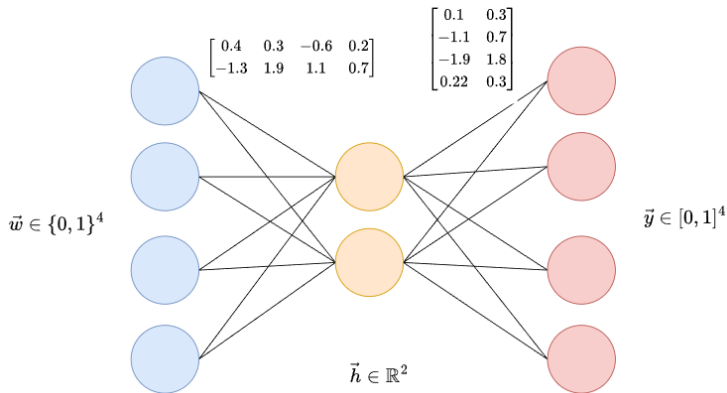
Important: In word2vec, the columns of the weight matrix $W_{wh} \in \mathbb{R}^{D \times V}$ are our *center-word* embeddings, and the rows of $W_{hy} \in \mathbb{R}^{D \times V}$ are our *context-word* embeddings.

So our task of predicting the probabilities was just a means of obtaining good weight matrices.

Suppose our vocab is “it”, “was”, “good”, “bad”. So word 1 in the vocabulary is “it”, word 2 is “was”, etc...

$$\begin{array}{l} \text{“it”} \rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{“was”} \rightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{“good”} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \text{“bad”} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \end{array}$$

Suppose we have trained a model with a vocabulary of size 4 and an embedding size of 2:



Using the weight matrices from the previous slide, our center word embeddings for each word are the vectors $\vec{e}_1, \dots, \vec{e}_4$, which correspond to the columns of W_{wh} :

$$W_{wh} = \begin{bmatrix} | & | & | & | \\ \vec{e}_1 & \vec{e}_2 & \vec{e}_3 & \vec{e}_4 \\ | & | & | & | \end{bmatrix} = \begin{bmatrix} 0.4 & 0.3 & -0.6 & 0.2 \\ -1.3 & 1.9 & 1.1 & 0.7 \end{bmatrix}$$

Using the weight matrices from the previous slide, our center word embeddings for each word are the vectors $\vec{e}_1, \dots, \vec{e}_4$, which correspond to the columns of W_{wh} :

$$W_{wh} = \begin{bmatrix} | & | & | & | \\ \vec{e}_1 & \vec{e}_2 & \vec{e}_3 & \vec{e}_4 \\ | & | & | & | \end{bmatrix} = \begin{bmatrix} 0.4 & 0.3 & -0.6 & 0.2 \\ -1.3 & 1.9 & 1.1 & 0.7 \end{bmatrix}$$

So our *center word* embeddings are:

$$\text{“it”} \rightarrow \begin{bmatrix} 0.4 \\ -1.3 \end{bmatrix} \quad \text{“was”} \rightarrow \begin{bmatrix} 0.3 \\ 1.9 \end{bmatrix} \quad \text{“good”} \rightarrow \begin{bmatrix} -0.6 \\ 1.1 \end{bmatrix} \quad \text{“bad”} \rightarrow \begin{bmatrix} 0.2 \\ 0.7 \end{bmatrix}$$

Our context word embeddings $\vec{c}_1, \dots, \vec{c}_4$ will be the rows of W_{hy} :

$$W_{hy} = \begin{bmatrix} -\vec{c}_1^T - \\ -\vec{c}_2^T - \\ -\vec{c}_3^T - \\ -\vec{c}_4^T - \end{bmatrix} = \begin{bmatrix} 0.1 & 0.3 \\ -1.1 & 0.7 \\ -1.9 & 1.8 \\ 0.22 & 0.3 \end{bmatrix}$$

Our context word embeddings $\vec{c}_1, \dots, \vec{c}_4$ will be the rows of W_{hy} :

$$W_{hy} = \begin{bmatrix} -\vec{c}_1^T - \\ -\vec{c}_2^T - \\ -\vec{c}_3^T - \\ -\vec{c}_4^T - \end{bmatrix} = \begin{bmatrix} 0.1 & 0.3 \\ -1.1 & 0.7 \\ -1.9 & 1.8 \\ 0.22 & 0.3 \end{bmatrix}$$

Our context word embeddings are:

$$\text{"it"} \rightarrow \begin{bmatrix} 0.1 \\ 0.3 \end{bmatrix} \quad \text{"was"} \rightarrow \begin{bmatrix} -1.1 \\ 0.7 \end{bmatrix} \quad \text{"good"} \rightarrow \begin{bmatrix} -1.9 \\ 1.8 \end{bmatrix} \quad \text{"bad"} \rightarrow \begin{bmatrix} 0.22 \\ 0.3 \end{bmatrix}$$

Our context word embeddings $\vec{c}_1, \dots, \vec{c}_4$ will be the rows of W_{hy} :

$$W_{hy} = \begin{bmatrix} -\vec{c}_1^T - \\ -\vec{c}_2^T - \\ -\vec{c}_3^T - \\ -\vec{c}_4^T - \end{bmatrix} = \begin{bmatrix} 0.1 & 0.3 \\ -1.1 & 0.7 \\ -1.9 & 1.8 \\ 0.22 & 0.3 \end{bmatrix}$$

Our context word embeddings are:

$$\text{“it”} \rightarrow \begin{bmatrix} 0.1 \\ 0.3 \end{bmatrix} \quad \text{“was”} \rightarrow \begin{bmatrix} -1.1 \\ 0.7 \end{bmatrix} \quad \text{“good”} \rightarrow \begin{bmatrix} -1.9 \\ 1.8 \end{bmatrix} \quad \text{“bad”} \rightarrow \begin{bmatrix} 0.22 \\ 0.3 \end{bmatrix}$$

Notice what happens when you take the dot product of a matrix with a one-hot vector:

$$\begin{bmatrix} 0.4 & 0.3 & -0.6 & 0.2 \\ -1.3 & 1.9 & 1.1 & 0.7 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \cdot 0.4 + 1 \cdot 0.3 + 0 \cdot -0.6 + 0 \cdot 0.2 \\ 0 \cdot -1.3 + 1 \cdot 1.9 + 0 \cdot 1.1 + 0 \cdot 0.7 \end{bmatrix} = \begin{bmatrix} 0.3 \\ 1.9 \end{bmatrix}$$

If the 1 is in position i , then it isolates column i of the matrix!

This gives us a new interpretation of what our network is doing!

Let's revisit our equations from earlier:

$$\vec{h}_i = W_{wh} \vec{w}_i \in \mathbb{R}^D$$

$\vec{w}_i$ is a one-hot vector with zeros everywhere except a 1 at element i .

Above we saw that $W_{wh} \vec{w}_i$ will simply isolate the values of column i of W_{wh} :

$$\vec{h}_i = W_{wh} \vec{w}_i = \begin{bmatrix} | & \dots & | \\ \vec{e}_1 & \dots & \vec{e}_V \\ | & \dots & | \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} = W_{wh}[:, i] = \vec{e}_i$$

This tells us that, when we provide our network with the one-hot encoding for word i , the value of $\vec{h}$ will simply be the center word embedding for word i !

Remember also that the row j of W_{hy} are the context word embedding of word j .
 Thus, if our original input was $\vec{w}_i$ then

$$\vec{u}_i = W_{hy} \vec{h}_i = W_{hy} (W_{hh} \vec{w}_i) = \begin{bmatrix} -\vec{c}_1^T - \\ \vdots \\ -\vec{c}_V^T - \end{bmatrix} \vec{e}_i = \begin{bmatrix} \vec{c}_1^T \cdot \vec{e}_i \\ \vdots \\ \vec{c}_V^T \cdot \vec{e}_i \end{bmatrix} = \begin{bmatrix} \text{dp-sim}(\vec{c}_1, \vec{e}_i) \\ \vdots \\ \text{dp-sim}(\vec{c}_V, \vec{e}_i) \end{bmatrix}$$

Remember also that the row j of W_{hy} are the context word embedding of word j . Thus, if our original input was $\vec{w}_i$ then

$$\vec{u}_i = W_{hy} \vec{h}_i = W_{hy} (W_{hh} \vec{w}_i) = \begin{bmatrix} -\vec{c}_1^T - \\ \vdots \\ -\vec{c}_V^T - \end{bmatrix} \vec{e}_i = \begin{bmatrix} \vec{c}_1^T \cdot \vec{e}_i \\ \vdots \\ \vec{c}_V^T \cdot \vec{e}_i \end{bmatrix} = \begin{bmatrix} \text{dp-sim}(\vec{c}_1, \vec{e}_i) \\ \vdots \\ \text{dp-sim}(\vec{c}_V, \vec{e}_i) \end{bmatrix}$$

Thus, each element of $\vec{u}_i$ is the dot product between the context word embedding for each vocabulary word, $\vec{c}_j$, and the center word embedding for each word.

Remember from earlier that the dot-product can be used to measure the similarity between two vectors!

Thus, we can re-interpret what our model is doing.

Given the one-hot encoding for word i , which we will call $\vec{w}_i$:

1. Compute the center-word embedding of word i as $\vec{e}_i = W_{wh}\vec{w}_i$
2. Compute the vector $\vec{u}_i$ where element j is the dot-product similarity between the context-word embedding of j and the center-word embedding of i
3. Feed $\vec{u}_i$ to softmax function. This will give higher probabilities to pairs i, j with higher dot-product similarities.

Thus, we can re-interpret what our model is doing.

Given the one-hot encoding for word i , which we will call $\vec{w}_i$:

1. Compute the center-word embedding of word i as $\vec{e}_i = W_{wh}\vec{w}_i$
2. Compute the vector $\vec{u}_i$ where element j is the dot-product similarity between the context-word embedding of j and the center-word embedding of i
3. Feed $\vec{u}_i$ to softmax function. This will give higher probabilities to pairs i, j with higher dot-product similarities.

Key Takeaway: Our model assigns higher probabilities to similar pairs of center and context embeddings.

training - visualized

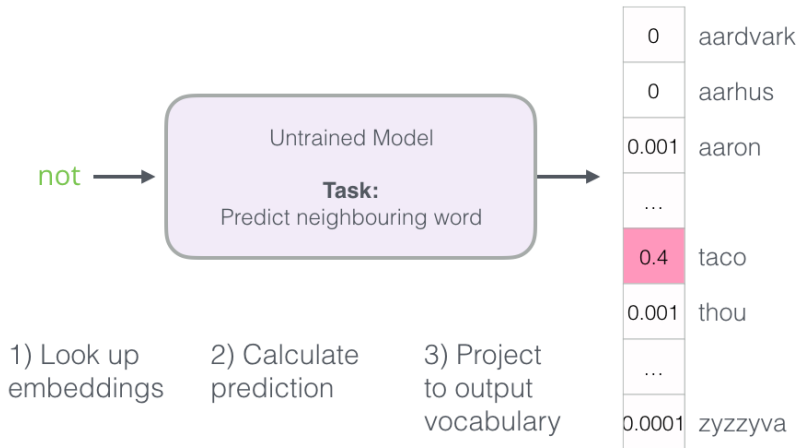
Let's abstract away the interpretation from earlier and just look at training:

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness

not →



training - visualized



training - visualized

Note that our targets are also one-hot encodings:

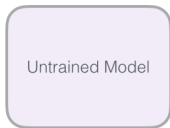
Actual Target		Model Prediction		Error
0		0 aardvark		0
0		0 aarhus		0
0		0.001 aaron		-0.001
...		...		...
0		0.4 taco		-0.4
1	-	0.001 thou	=	0.999
...		...		...
0		0.0001 zyzyva		-0.0001

training - visualized

Actual
Target

0
0
0
...
0
1
...
0

not



Model
Prediction

0	aardvark
0	aarhus
0.001	aaron
...	...
0.4	taco
0.001	thou
...	...
0.0001	zyzzyva

Error

0
0
-0.001
...
-0.4
0.999
...
-0.0001

Update
Model
Parameters

As we said before, our model weights *are* the embeddings, so a weight update can simply be thought of as updating our embeddings.

Problem: W_{hy} can be a really big matrix (eg. if vocab has 10,000 words and embedding dimension is 1024 then it contains 10,240,000 elements). Currently, we are updating all embeddings at each time step. This will make training very slow. Can we make this more efficient?

Answer: Yes, but we will need to reformulate our task a bit...

small change to task

We will use the idea of **negative sampling**.

At each time step we will only update $1 + K < V$ embeddings.

For each center word $\vec{v}$, we will update the embeddings for:

- 1 context word that is a neighbour of $\vec{v}$
- K random context words that are *not* neighbours of $\vec{v}$

For each context word, the network predicts the probability that it's a neighbour of $\vec{v}$.

Note that this is a yes or no question, so we are now performing *binary classification*, where we will apply sigmoid in the output.

negative sampling

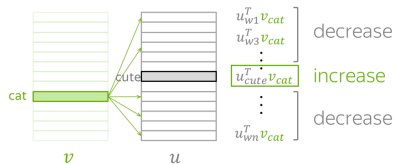
Dot product of v_{cat} :

- with u_{cute} - increase,
- with all other u - decrease



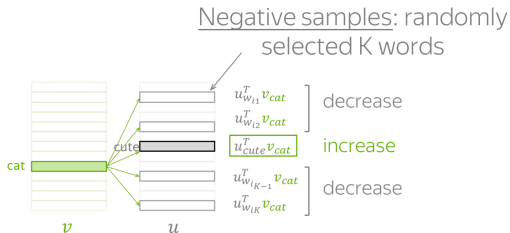
Dot product of v_{cat} :

- with u_{cute} - increase,
- with a subset of other u - decrease



Parameters to be updated:

- v_{cat}
 - u_w for all w in the vocabulary
- $|V| + 1$ vectors



Parameters to be updated:

- v_{cat}
 - u_{cute} and u_w for w in K negative examples
- $K + 2$ vectors

negative sampling

Let's see how the dataset changes:

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	make	1

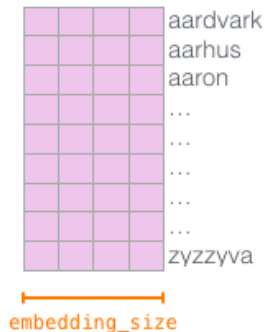
Pick randomly from vocabulary
(random sampling)

Word	Count	Probability
aardvark		
aarhus		
aaron		
taco		
thou		
zyzzyva		

Embedding



Context



Embedding

				aardvark
				aarhus
				aaron
				...
				not
				...
				...
				...
				zyzzyva

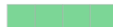
Context

				aardvark
				aarhus
				aaron
				...
				taco
				...
				thou
				...
				zyzzyva

Look up
embeddings



not



aaron



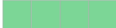
taco

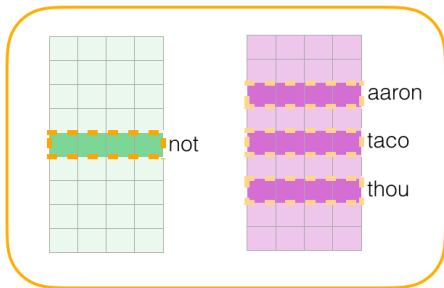


thou



input word	output word	target	input • output	sigmoid()	Error
not 	thou 	1	0.2	0.55	0.45
not 	aaron 	0	-1.11	0.25	-0.25
not 	taco 	0	0.74	0.68	-0.68

input word	output word	target	input • output	sigmoid()	Error
not 	thou 	1	0.2	0.55	0.45
not 	aaron 	0	-1.11	0.25	-0.25
not 	taco 	0	0.74	0.68	-0.68



**Update
Model
Parameters**

Once again, when training is done, we can now look up our embeddings in the center-word matrix (green).

ELMo

We haven't yet addressed **polysemy**.

broke { I **broke** the vase
dawn **broke**
he **broke** the silence
the news **broke**
the machine **broke**
she **broke** the world record
he **broke** the law

ELMo

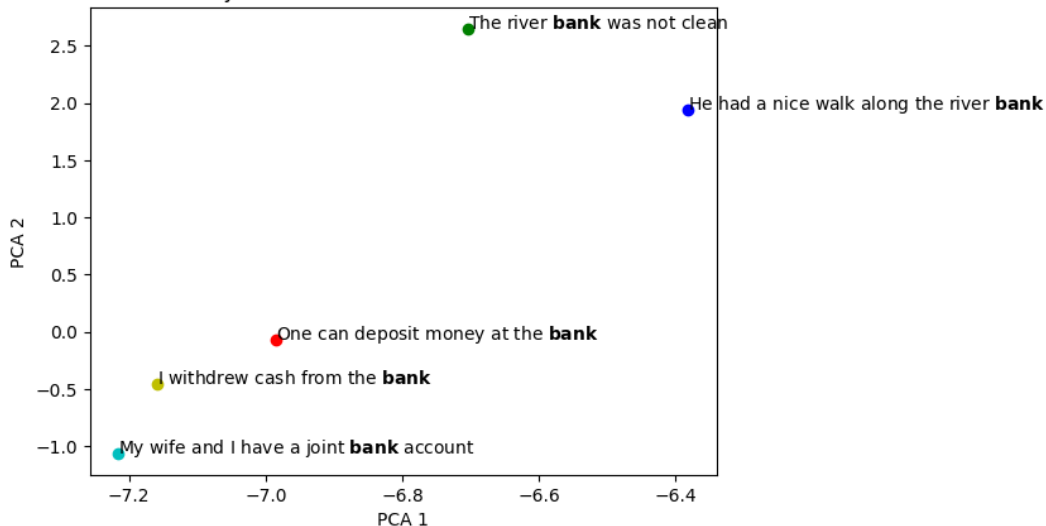
We haven't yet addressed **polysemy**.

broke { I **broke** the vase
dawn **broke**
he **broke** the silence
the news **broke**
the machine **broke**
she **broke** the world record
he **broke** the law

Can we embed context? We can, using the **ELMo** model and its **contextual vectors**.



Layer 1 ELMo vectors of the word bank



Note: The architecture for ELMo is quite complex. The key takeaways here should be about the process rather than the specific details.

Our strategy:

1. ELMo is a bi-directional two-layer LSTM. We will train it on the task of next token prediction.

Our strategy:

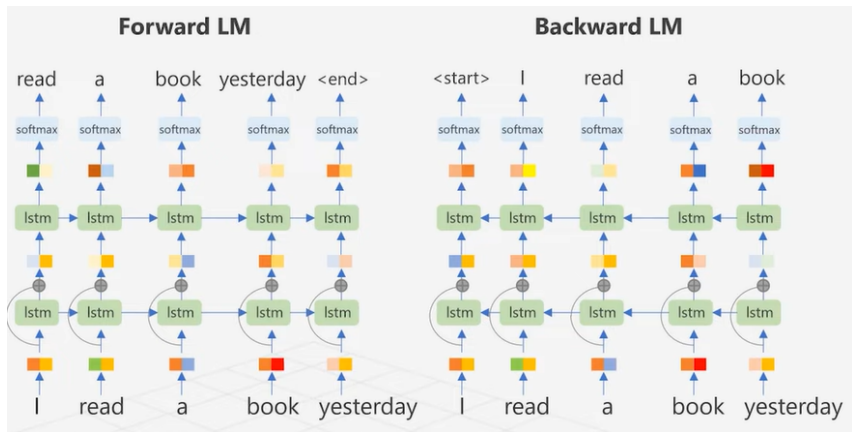
1. ELMo is a bi-directional two-layer LSTM. We will train it on the task of next token prediction.
2. Once ELMo is trained, we can feed it a sentence and it will produce an embedding of each word that depends on the *context*.

Our strategy:

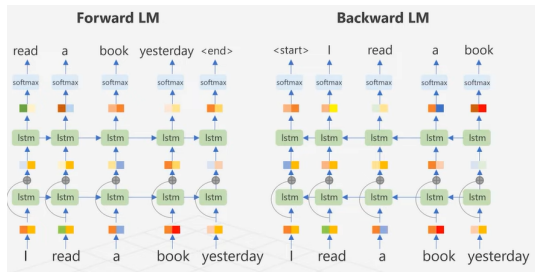
1. ELMo is a bi-directional two-layer LSTM. We will train it on the task of next token prediction.
2. Once ELMo is trained, we can feed it a sentence and it will produce an embedding of each word that depends on the *context*.
3. We now have embeddings that we can feed into another network.

architecture

ELMo is a bi-directional LSTM with two layers:



architecture

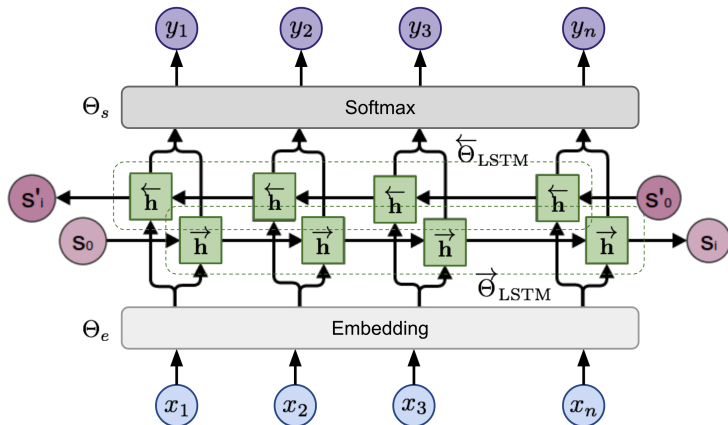


We do next token prediction where we predict the probabilities of the next word. We can imagine the forward and backwards LMs as unrelated networks with different tasks:

- In the forward LM, we predict the next word in the future.
- In the backwards LM, we predict the next word in the past.

The loss function takes into account the accuracy of the predictions in the forward and backwards directions.

The networks aren't actually unrelated since the the input and output layers of the forward and backwards LM share weights (denoted Θ_e and Θ_s here):

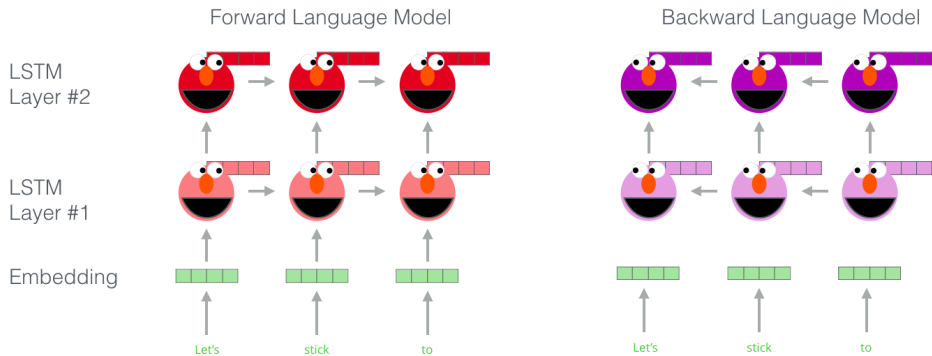


Now assume the network is *trained*. Let's see how we can use it to create word embeddings that include context.

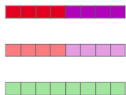
Here we will look at how we might embed the word “stick” in

“let's stick to improvisation in this skit”

We feed the entire sentence into the forward and backward LMs:



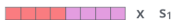
1- Concatenate hidden layers



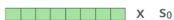
2- Multiply each vector by a weight based on the task



$\times s_2$



$\times s_1$



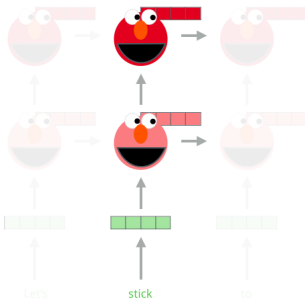
$\times s_0$

3- Sum the (now weighted) vectors

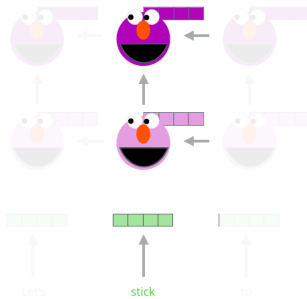


ELMo embedding of "stick" for this task in this context

Forward Language Model



Backward Language Model



The weights s_0 , s_1 , s_2 will be learned as a part of the downstream task (ie. the task we are using these embeddings for). Crucially, in the downstream task, the LSTM part is already trained and its weights are frozen (ie. they aren't updated).

A small note

The attention mechanism we looked at last time came from the paper “Neural Machine Translation by Jointly Learning to Align and Translate” by Dzmitry Bahdanau, KyungHyun Cho, and Yoshua Bengio. The last two authors are from UdeM!

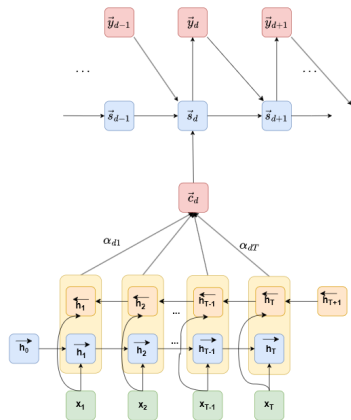
Yoshua Bengio



- Went to McGill for his BS, MSc, and PhD!
- Founded Mila
- Winner of the Turing Prize (basically Nobel prize for CS)
- Known as one of the “Godfathers of AI”

attention (review)

additive attention (review)



Recall “additive attention” which we covered last week.

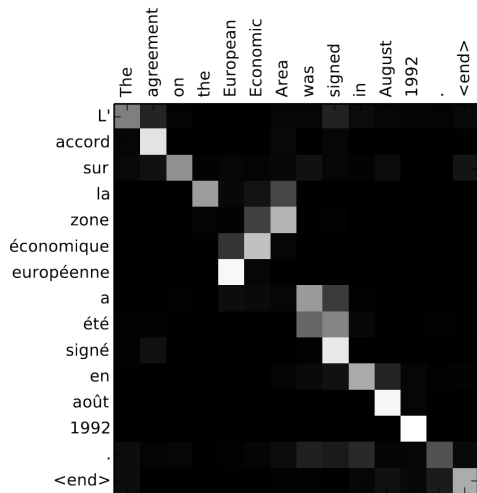
$$\vec{s}_i = f(\vec{s}_{i-1}, \vec{y}_{i-1}, \vec{c}_i)$$

$$\vec{c}_i = \sum_{j=1}^T \alpha_{ij} \mathbf{h}_j$$

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^T \exp(e_{ik})}$$

$$e_{ij} = \text{score}(\vec{s}_{i-1}, \mathbf{h}_j)$$

$$\text{score}(\vec{s}_{i-1}, \mathbf{h}_j) = \vec{v} \cdot \tanh(W_a[\vec{s}_{i-1}; \mathbf{h}_j])$$



This is the matrix α , where α_{ij} is the weight of encoder state $\mathbf{h}_j$ in the computation

$$\vec{c}_i = \sum_{j=1}^T \alpha_{ij} \mathbf{h}_j$$

The key takeaway of this mechanism was that the input to the encoder, $\vec{c}_i$ is a weighted sum of the hidden states of the decoder, where encoder states with similarity scores get larger weights.

Our scoring function was:

$$\text{score}(\vec{s}_{i-1}, \mathbf{h}_j) = \vec{v} \cdot \tanh(W_a[\vec{s}_{i-1}; \mathbf{h}_j])$$

But we could have used any scoring function...

additive attention (review)

Example

Earlier we saw dot-product being used to compute similarity:

$$\text{dp-sim}(\vec{x}, \vec{y}) = \vec{x}^T \cdot \vec{y}$$

self-attention

We now introduce the idea of “self-attention”:

Encoder-decoder attention looks:

- **from:** one current decoder state
- **to:** all encoder states

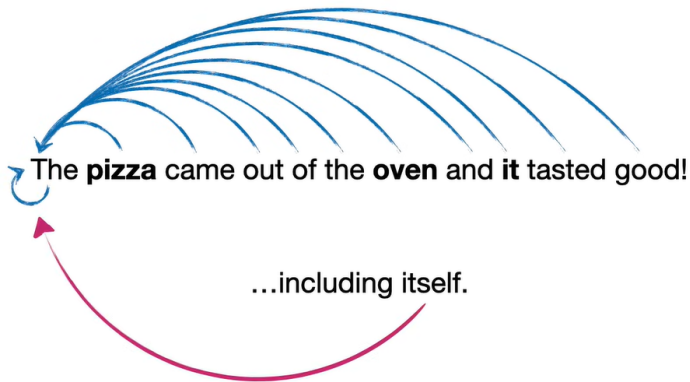
Self-attention looks at:

- **from:** each state from a set of states
- **to:** each state from the *same* set of states

self-attention

Example

Given a sentence, self-attention will compute the similarities between each pair of words in the sentence:



self-attention

At a high-level, we will take as input a set of inputs (vectors) and compute a new set of vectors that describe how related each of the input vectors are to each other.

Mathematically, the inputs and outputs of self-attention are:

- Input: A set of d_{in} -dimensional input vectors $\{\vec{x}_1, \dots, \vec{x}_T\}$
- Output: A set of d_v -dimensional vectors $\{\vec{z}_1, \dots, \vec{z}_T\}$ where $\vec{z}_i$ contains information about how $\vec{x}_i$ relates to all the other input vectors $\vec{x}_j$

self-attention

At a high-level, we will take as input a set of inputs (vectors) and compute a new set of vectors that describe how related each of the input vectors are to each other.

Mathematically, the inputs and outputs of self-attention are:

- Input: A set of d_{in} -dimensional input vectors $\{\vec{x}_1, \dots, \vec{x}_T\}$
- Output: A set of d_v -dimensional vectors $\{\vec{z}_1, \dots, \vec{z}_T\}$ where $\vec{z}_i$ contains information about how $\vec{x}_i$ relates to all the other input vectors $\vec{x}_j$

Question: “how do we compute $\vec{z}_i$?”

We will compute $\vec{z}_i \in \mathbb{R}^{d_v}$ as a weighted sum of *value vectors*, denoted $\vec{v}_j$:

$$\vec{z}_i = \sum_{j=1}^T \alpha_{ij} \vec{v}_j$$

We will compute $\vec{z}_i \in \mathbb{R}^{d_v}$ as a weighted sum of *value vectors*, denoted $\vec{v}_j$:

$$\vec{z}_i = \sum_{j=1}^T \alpha_{ij} \vec{v}_j$$

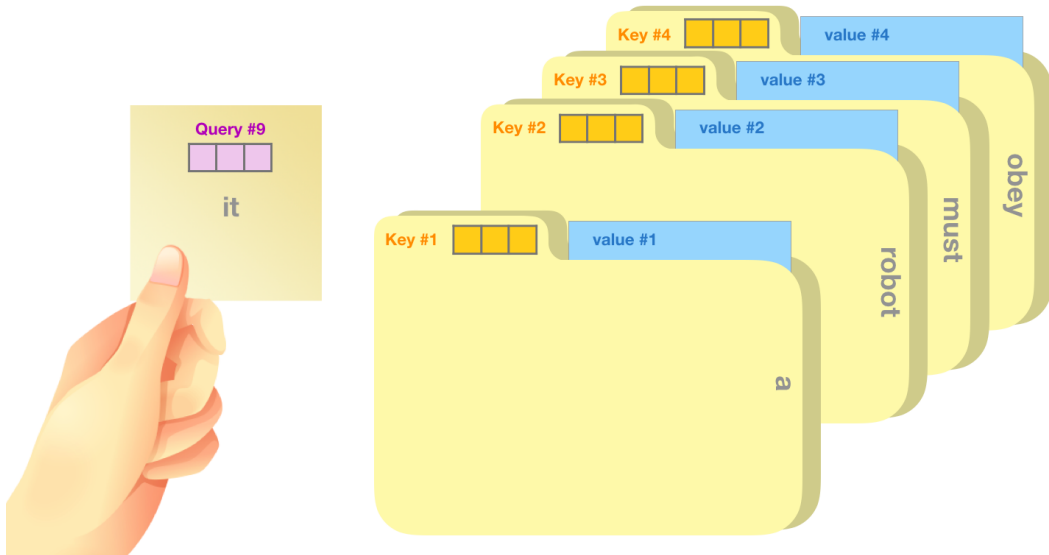
Look familiar?

We will compute $\vec{z}_i \in \mathbb{R}^{d_v}$ as a weighted sum of *value vectors*, denoted $\vec{v}_j$:

$$\vec{z}_i = \sum_{j=1}^T \alpha_{ij} \vec{v}_j$$

Look familiar?

Question: How do we compute α_{ij} and $\vec{v}_j$?



Compute the key, query, and value vectors for the word embeddings.

$$\vec{k}_i = W_K \vec{x}_i \in \mathbb{R}^{d_k} \quad \vec{q}_i = W_Q \vec{x}_i \in \mathbb{R}^{d_k} \quad \vec{v}_i = W_V \vec{x}_i \in \mathbb{R}^{d_v}$$

These are weight matrices that our model will *learn*.

Compute the key, query, and value vectors for the word embeddings.

$$\vec{k}_i = W_K \vec{x}_i \in \mathbb{R}^{d_k} \quad \vec{q}_i = W_Q \vec{x}_i \in \mathbb{R}^{d_k} \quad \vec{v}_i = W_V \vec{x}_i \in \mathbb{R}^{d_v}$$

These are weight matrices that our model will *learn*.

Using these, we compute the interaction between word i and j , and then the attention score.

$$e_{ij} = \text{score}(\vec{q}_i, \vec{k}_j) = \frac{\vec{q}_i \cdot \vec{k}_j}{\sqrt{d_k}} \qquad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^T \exp(e_{ik})}$$

- e_{ij} tells us “how related is input j to input i ?”
- α_{ij} are the scores from e_{ij} turned into weights

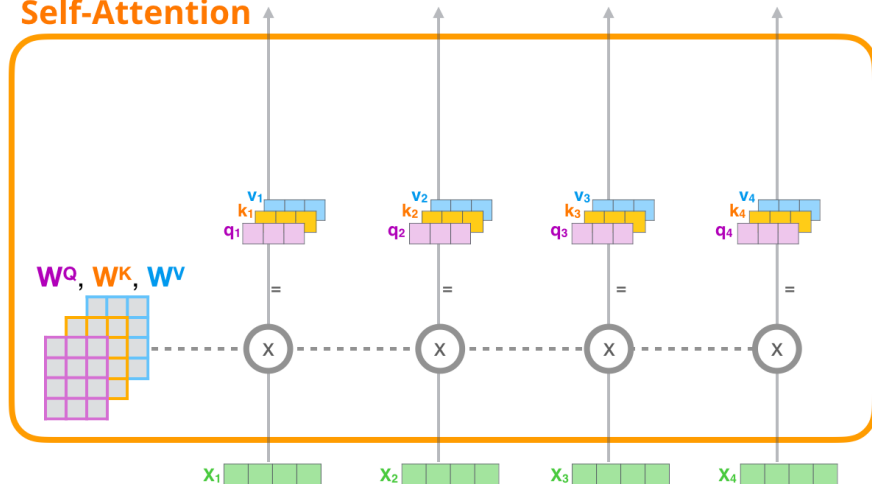
Putting it all together, we compute $\vec{z}_i$ with the following steps:

1. For each input $\vec{x}_i$, compute $\vec{k}_i = W_K \vec{x}_i$, $\vec{q}_i = W_Q \vec{x}_i$, $\vec{v}_i = W_V \vec{x}_i$
2. For each i , compute the similarity of its query vector $\vec{q}_i$ to all the other key vectors $\vec{k}_j$ to compute similarities $e_{ij} = \text{score}(\vec{q}_i, \vec{k}_j)$
3. Turn each the vector $\vec{e}_i = [e_{i1}, e_{i2}, \dots, e_{iT}]$ into a weight vector $\vec{\alpha}_i = [\alpha_{i1}, \alpha_{i2}, \dots, \alpha_{iT}] = \text{softmax}(\vec{e}_i)$
4. For each i , compute $\vec{z}_i$ as $\vec{z}_i = \sum_{j=1}^T \alpha_{ij} \vec{v}_j$

Key Takeaway: The computation of $\vec{z}_i$ is a weighted sum of value vectors, where value vectors of more relevant words get a higher weight!

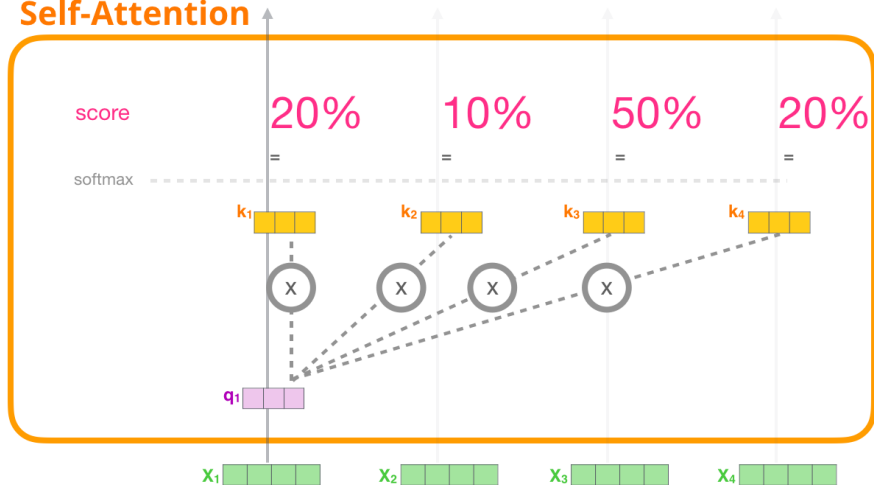
1) For each input token, create a **query vector**, a **key vector**, and a **value vector** by multiplying by weight Matrices W^Q , W^K , W^V

Self-Attention



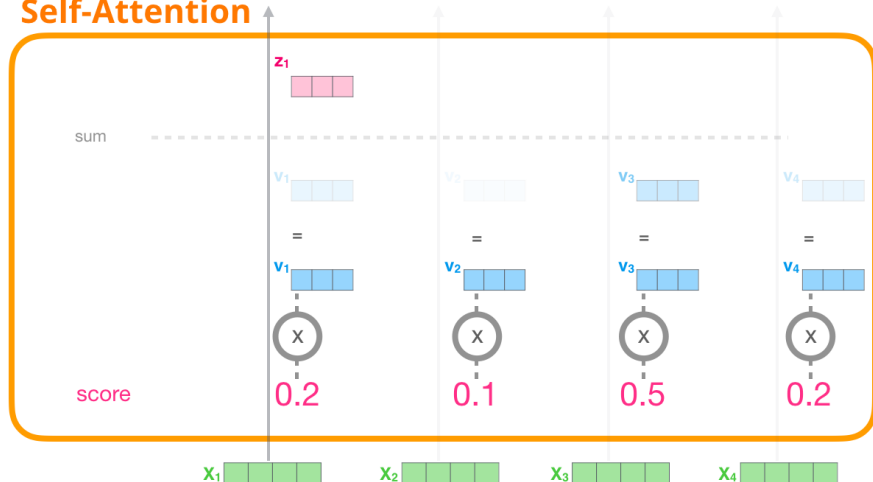
2) Multiply (dot product) the current **query vector**, by all the **key vectors**, to get a score of how well they match

Self-Attention



3) Multiply the **value vectors** by the **scores**, then sum up

Self-Attention



We can actually make this computation a lot more concise using matrix operations! This is super common in machine learning and helps speed up computations since matrix operations can be done in parallel.

To start, we will stack all of our inputs into a big matrix where each row is an input:

$$X = \begin{bmatrix} -\vec{x}_1^T - \\ \vdots \\ -\vec{x}_T^T - \end{bmatrix} \in \mathbb{R}^{T \times d_{\text{in}}}$$

To start, we will stack all of our inputs into a big matrix where each row is an input:

$$X = \begin{bmatrix} -\vec{x}_1^T - \\ \vdots \\ -\vec{x}_T^T - \end{bmatrix} \in \mathbb{R}^{T \times d_{\text{in}}}$$

Now we can compute all of our key vectors in a single matrix multiplication:

$$K = X W_K = \begin{bmatrix} -\vec{x}_1^T - \\ \vdots \\ -\vec{x}_T^T - \end{bmatrix} W_K = \begin{bmatrix} -\vec{k}_1^T - \\ \vdots \\ -\vec{k}_T^T - \end{bmatrix} \in \mathbb{R}^{T \times d_k}$$

where $W_K \in \mathbb{R}^{d_{\text{in}} \times d_k}$

Similarly, we can compute all our query and value vectors in a single matrix multiplication:

$$Q = X W_Q = \begin{bmatrix} -\vec{x}_1^T - \\ \vdots \\ -\vec{x}_T^T - \end{bmatrix} W_Q = \begin{bmatrix} -\vec{q}_1^T - \\ \vdots \\ -\vec{q}_T^T - \end{bmatrix} \in \mathbb{R}^{T \times d_k}$$

where $W_Q \in \mathbb{R}^{d_{\text{in}} \times d_k}$

$$V = X W_V = \begin{bmatrix} -\vec{x}_1^T - \\ \vdots \\ -\vec{x}_T^T - \end{bmatrix} W_V = \begin{bmatrix} -\vec{v}_1^T - \\ \vdots \\ -\vec{v}_T^T - \end{bmatrix} \in \mathbb{R}^{T \times d_v}$$

where $W_V \in \mathbb{R}^{d_{\text{in}} \times d_v}$

To compute the similarities we can also use matrix multiplication!

$$\mathbf{e} = \frac{1}{\sqrt{d_k}} \mathbf{Q} \mathbf{K}^T = \frac{1}{\sqrt{d_k}} \begin{bmatrix} -\vec{q}_1^T \\ \vdots \\ -\vec{q}_T^T \end{bmatrix} \begin{bmatrix} | & & | \\ \vec{k}_1 & \dots & \vec{k}_T \\ | & & | \end{bmatrix} \in \mathbb{R}^{T \times T}$$

which gives you a matrix where $e_{ij} = \frac{\vec{q}_i \cdot \vec{k}_j}{\sqrt{d_k}}$ and row i is the vector $\vec{e}_i = [e_{i1}, e_{i2}, \dots, e_{iT}]$

To compute the similarities we can also use matrix multiplication!

$$\mathbf{e} = \frac{1}{\sqrt{d_k}} \mathbf{Q} \mathbf{K}^T = \frac{1}{\sqrt{d_k}} \begin{bmatrix} -\vec{q}_1^T \\ \vdots \\ -\vec{q}_T^T \end{bmatrix} \begin{bmatrix} | & & | \\ \vec{k}_1 & \dots & \vec{k}_T \\ | & & | \end{bmatrix} \in \mathbb{R}^{T \times T}$$

which gives you a matrix where $e_{ij} = \frac{\vec{q}_i \cdot \vec{k}_j}{\sqrt{d_k}}$ and row i is the vector $\vec{e}_i = [e_{i1}, e_{i2}, \dots, e_{iT}]$

We can also compute the scores by taking the softmax of each row:

$$\boldsymbol{\alpha} = \begin{bmatrix} -\text{softmax}(\vec{e}_1) \\ \vdots \\ -\text{softmax}(\vec{e}_T) \end{bmatrix} = \begin{bmatrix} \alpha_{11} & \dots & \alpha_{1T} \\ \vdots & \ddots & \vdots \\ \alpha_{T1} & \dots & \alpha_{TT} \end{bmatrix} \in [0, 1]^{T \times T}$$

Finally, we can compute a matrix where row i is z_i :

$$Z = \alpha V = \begin{bmatrix} \alpha_{11} & \dots & \alpha_{1T} \\ \vdots & \ddots & \vdots \\ \alpha_{T1} & \dots & \alpha_{TT} \end{bmatrix} \begin{bmatrix} -\vec{v}_1^T - \\ \vdots \\ -\vec{v}_T^T - \end{bmatrix} = \begin{bmatrix} -\vec{z}_1^T - \\ \vdots \\ -\vec{z}_T^T - \end{bmatrix} \in \mathbb{R}^{T \times d_v}$$

Note: If you aren't used to working with matrix multiplication, this last result might not seem so obvious. Don't worry!

Let's see how our old steps and see how they changed:

Old

1. For each input $\vec{x}_i$, compute $\vec{k}_i = W_K \vec{x}_i$, $\vec{q}_i = W_Q \vec{x}_i$, $\vec{v}_i = W_V \vec{x}_i$
2. For each i , compute the similarity of its query vector $\vec{q}_i$ to all the other key vectors $\vec{k}_j$ to compute similarities $e_{ij} = \text{score}(\vec{q}_i, \vec{k}_j)$
3. Turn each the vector $\vec{e}_i = [e_{i1}, e_{i2}, \dots, e_{iT}]$ into a weight vector $\vec{\alpha}_i = [\alpha_{11}, \alpha_{12}, \dots, \alpha_{1T}] = \text{softmax}(\vec{e}_i)$
4. For each i , compute $\vec{z}_i$ as $\vec{z}_i = \sum_{j=1}^T \alpha_{ij} \vec{v}_j$

New

1. Compute matrices $K = XW_K$, $Q = XW_Q$, $V = XW_V$, where row i of each matrix gives you $\vec{k}_i$, $\vec{q}_i$, and $\vec{v}_i$ respectively
2. Compute matrix $e = \frac{1}{\sqrt{d_k}} QK^T$ and take the softmax of each row to get α
3. Compute $Z = \alpha V$, where row i is $\vec{z}_i$

Let's see some illustrations of this.

$$\mathbf{X} \times \mathbf{W}^Q = \mathbf{Q}$$


The diagram illustrates the matrix multiplication $\mathbf{X} \times \mathbf{W}^Q = \mathbf{Q}$. Matrix $\mathbf{X}$ is a 2x4 grid of green cells. Matrix $\mathbf{W}^Q$ is a 4x4 grid of purple cells, with the first two columns highlighted in a darker purple. Matrix $\mathbf{Q}$ is a 2x4 grid of purple cells. The multiplication is indicated by a 'x' symbol between the matrices and an '=' symbol before the result.

$$\mathbf{X} \times \mathbf{W}^K = \mathbf{K}$$


The diagram illustrates the matrix multiplication $\mathbf{X} \times \mathbf{W}^K = \mathbf{K}$. Matrix $\mathbf{X}$ is a 2x4 grid of green cells. Matrix $\mathbf{W}^K$ is a 4x4 grid of orange cells, with the first two columns highlighted in a darker orange. Matrix $\mathbf{K}$ is a 2x4 grid of orange cells. The multiplication is indicated by a 'x' symbol between the matrices and an '=' symbol before the result.

$$\mathbf{X} \times \mathbf{W}^V = \mathbf{V}$$


The diagram illustrates the matrix multiplication $\mathbf{X} \times \mathbf{W}^V = \mathbf{V}$. Matrix $\mathbf{X}$ is a 2x4 grid of green cells. Matrix $\mathbf{W}^V$ is a 4x4 grid of blue cells, with the first two columns highlighted in a darker blue. Matrix $\mathbf{V}$ is a 2x4 grid of blue cells. The multiplication is indicated by a 'x' symbol between the matrices and an '=' symbol before the result.

$$\text{softmax}\left(\frac{\begin{matrix} \text{Q} \\ \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array} \end{matrix} \times \begin{matrix} \text{K}^T \\ \begin{array}{|c|c|} \hline \square & \square \\ \hline \square & \square \\ \hline \square & \square \\ \hline \end{array} \end{matrix}}{\sqrt{d_k}}\right) \begin{matrix} \text{V} \\ \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array} \end{matrix}$$

$$= \begin{matrix} \text{Z} \\ \begin{array}{|c|c|c|} \hline \square & \square & \square \\ \hline \square & \square & \square \\ \hline \end{array} \end{matrix}$$

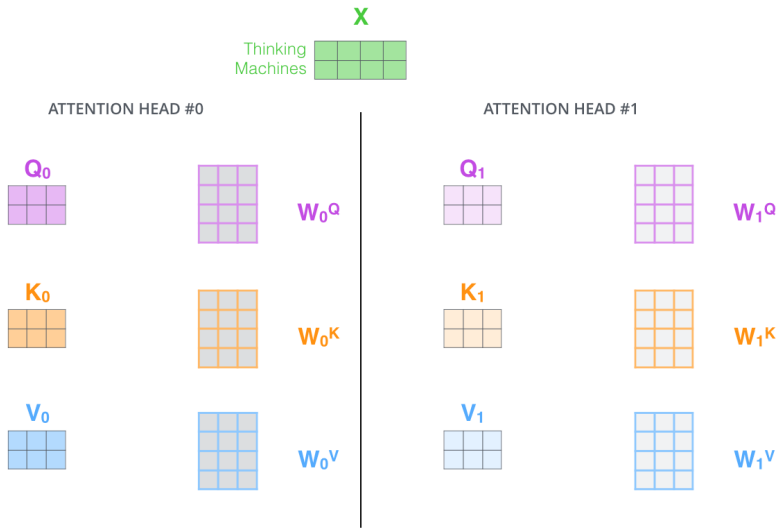
self-attention code

```
1 class SelfAttention(nn.Module):
2     def __init__(self, d_in, d_k, d_v):
3         super(SelfAttention, self).__init__()
4         self.d_in = d_in
5         self.d_k = d_k
6         self.d_v = d_v
7         self.W_query = nn.Parameter(torch.rand(d_in, d_k))
8         self.W_key = nn.Parameter(torch.rand(d_in, d_k))
9         self.W_value = nn.Parameter(torch.rand(d_in, d_v))
10    def forward(self, X):
11        Q = X @ self.W_query
12        K = X @ self.W_key
13        V = X @ self.W_value
14        e = Q @ K.T / math.sqrt(self.d_k)
15        alpha = F.softmax(attention_scores, dim=-1)
16        Z = attention_weights @ V
17        return Z
```

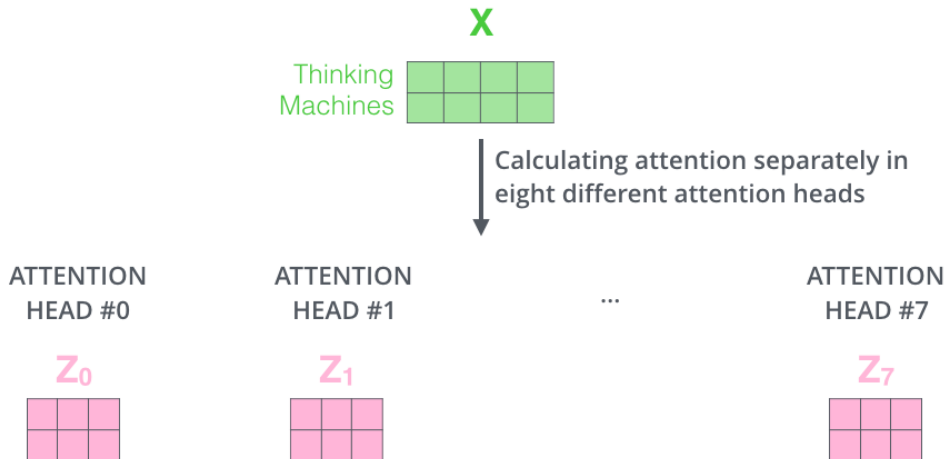
multi-headed attention

Multi-headed attention is an extension of the self-attention mechanism we just saw.

Each “head” takes the same X as input and performs self-attention. Here we have two attention heads:



We could have 8 heads, where each head computes its own Z :



We then concatenate then multiply by a huge weight matrix W^O to combine all the results together into a smaller Z where row i is $\vec{z}_i$:

1) This is our input sentence*

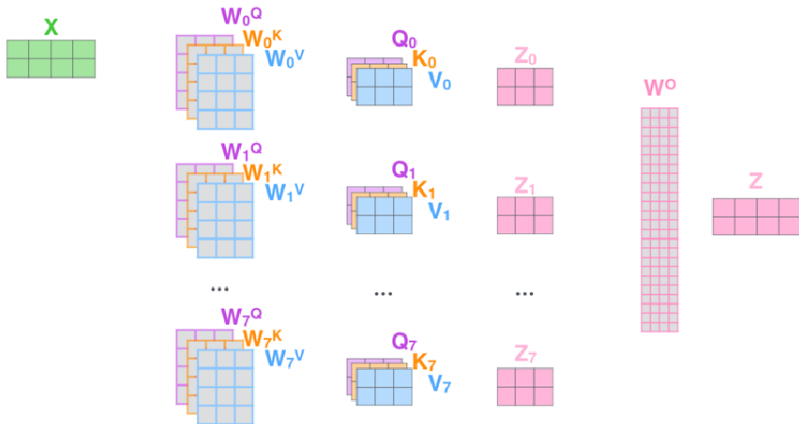
2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

4) Calculate attention using the resulting $Q/K/V$ matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

Thinking
Machines



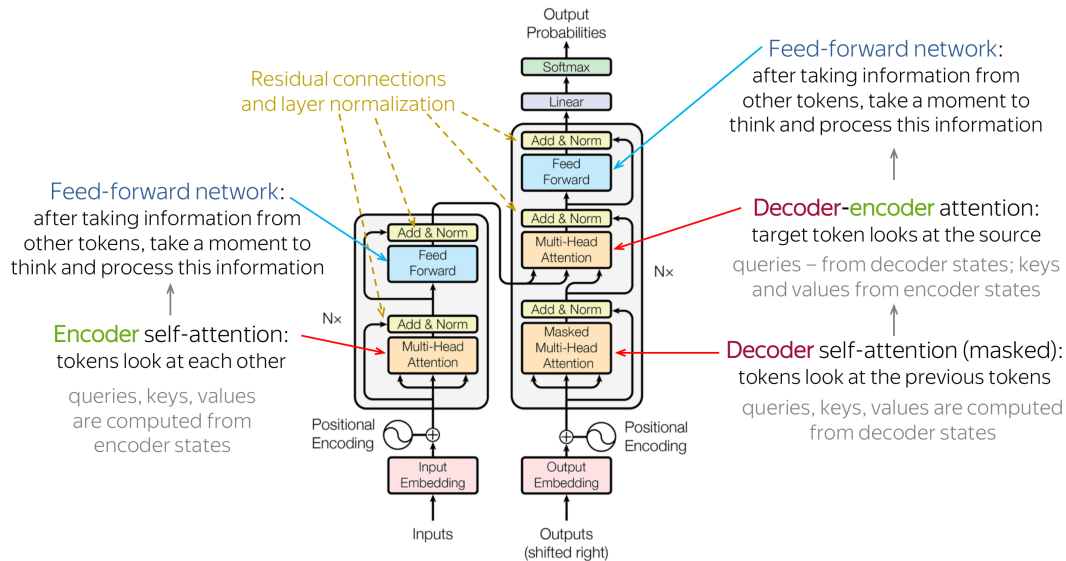
transformers

The self-attention mechanism is used in a very popular kind of model called a *transformer* introduced in the infamous “Attention is All You Need Paper” (Cited by **203,120**)

Transformers consist of two parts:

- An encoder
- A decoder

Left side is the encoder and right side is the decoder:



The full encoder-decoder model can be used for machine translation (eg. english to french). Let's contrast with the existing models we've seen so far:

	Seq2seq without attention	Seq2seq with attention	Transformer
processing within encoder	RNN/CNN	RNN/CNN	attention
processing within decoder	RNN/CNN	RNN/CNN	attention
decoder-encoder interaction	static fixed- sized vector	attention	attention

Depending on the task, you might only use one-half of the transformer:

- Encoder only: Text embeddings (eg. for retrieval augmented generation)
- Decoder only: ChatGPT
- Encoder-decoder: machine translation

We will focus on decoder-only models:

- Encoder-only: Text embeddings (eg. for retrieval augmented generation)
- **Decoder-only: ChatGPT**
- Encoder-decoder: machine translation

small note

One of the authors of the original GPT-1, GPT-2, and GPT-3 papers is Ilya Sutskever!

Ilya Sutskever



- Went to UofT for his BS, MSc, and PhD!
- Created the **seq2seq** algorithm
- Co-founder of OpenAI

We will look at an earlier version of ChatGPT, which is a fine-tuned version of GPT-3.

The model is trained in two distinct phase:

1. Pre-training: here the training objective is next token (word) prediction, which we have seen before
 - ▶ This phase teaches the model how language works (ie. “how to generate a realistic sentence”)
2. Fine-tuning: consists of multiple stages
 - ▶ This phase teaches the model how to be a good chat bot

We will look at an earlier version of ChatGPT, which is a fine-tuned version of GPT-3.

The model is trained in two distinct phase:

1. Pre-training: here the training objective is next token (word) prediction, which we have seen before
 - ▶ This phase teaches the model how language works (ie. “how to generate a realistic sentence”)
2. Fine-tuning: consists of multiple stages
 - ▶ This phase teaches the model how to be a good chat bot

We will start by looking at the *pre-training* phase

pre-training

Goal: Given the first k words in a sentence, predict the $(k + 1)$ -th word.

- Input: A sentence
- Output: Probability of next word for each word in the sentence

Note: Unlike RNNs which go one input at a time, transformers can make predictions about the *entire* input sequence at once!

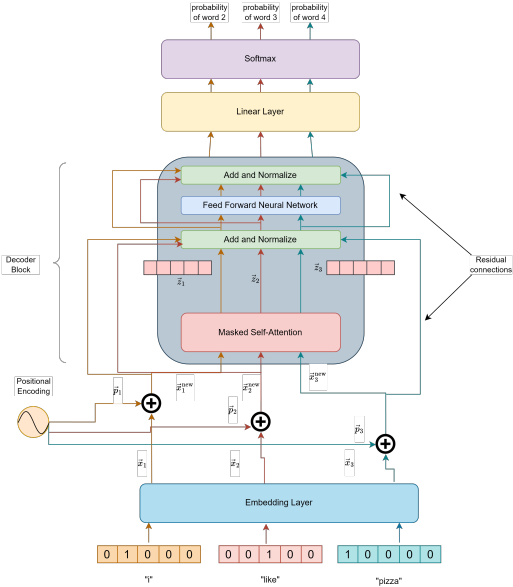
pre-training: dataset

Inputs						Outputs					
$\vec{w}_1$	$\vec{w}_2$	$\vec{w}_3$	$\vec{w}_4$	...	$\vec{w}_T$	$\vec{y}_1$	$\vec{y}_2$	$\vec{y}_3$	$\vec{y}_4$	...	$\vec{y}_T$
i	like	pizza	<END>	...	<EMPTY>	like	pizza	<END>	<EMPTY>	...	<EMPTY>
hello	world	<END>	<EMPTY>	...	<EMPTY>	world	<END>	<EMPTY>	<EMPTY>	...	<EMPTY>
transformers	are	the	model	...	paper	are	the	model	<EMPTY>	...	with

Notes

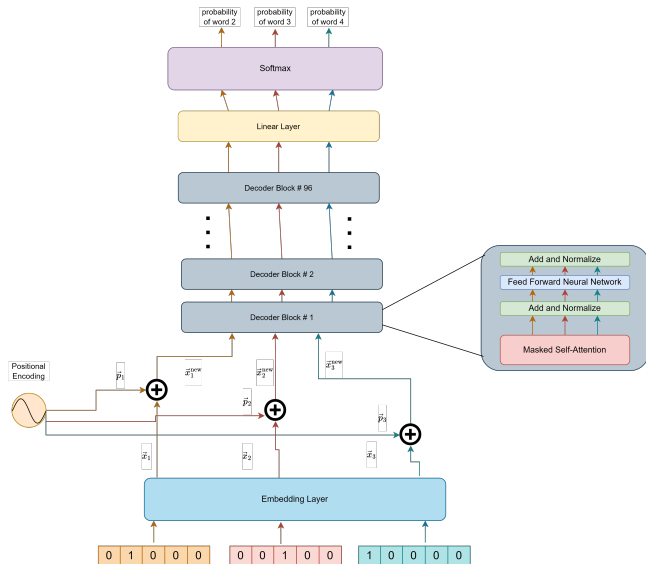
- Unlike RNNs, transformers have fixed size inputs and outputs (during training). The inputs thus have a maximum sequence length T (called the “context window”). In GPT-3 this is 2048 tokens. If the input is shorter, we can fill the rest of the sequence with “EMPTY ” tokens.
- Our inputs and outputs will be represented as one-hot encodings.

architecture



architecture

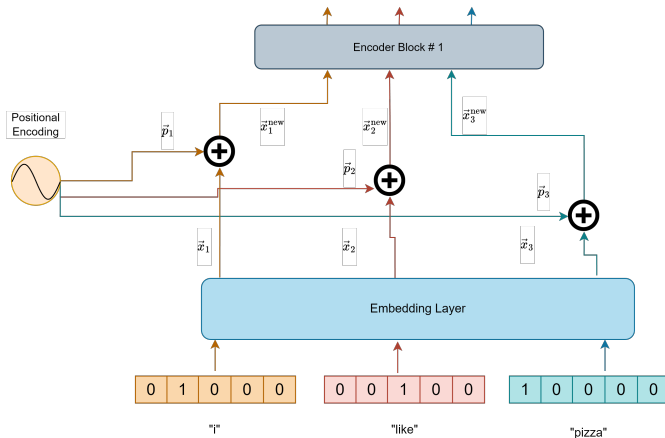
GPT-3 is just a stack of these 96 of these “decoder blocks”:



inputs

Let's zoom in on each part of the network.

In the input, we pass our one-hot encodings through a linear layer to get word embeddings, and then add a “positional encoding vector” (more on this soon):



positional encoding

One key detail we have overlooked so far is that self-attention works on *sets*. That is, it doesn't really consider the order of the inputs.

This is different from RNNs where we fed in each input of a sequence in order.

If we want to use this mechanism on sequences, we often first encode some information about the order of the inputs using a *positional encoding*.

If we want to represent a sequence $\vec{x}_1, \dots, \vec{x}_T$ where $\vec{x}_t \in \mathbb{R}^{d_{\text{in}}}$, then we can compute a new vector $\vec{x}_t^{\text{new}}$ where

$$\vec{x}_t^{\text{new}} = \vec{x}_t + \vec{p}_t$$

where

$$\vec{p}_t = \left[\sin\left(\frac{t}{10000^{2/d_{\text{in}}}}\right), \cos\left(\frac{t}{10000^{4/d_{\text{in}}}}\right), \sin\left(\frac{t}{10000^{6/d_{\text{in}}}}\right), \dots \right]$$

If we want to represent a sequence $\vec{x}_1, \dots, \vec{x}_T$ where $\vec{x}_t \in \mathbb{R}^{d_{\text{in}}}$, then we can compute a new vector $\vec{x}_t^{\text{new}}$ where

$$\vec{x}_t^{\text{new}} = \vec{x}_t + \vec{p}_t$$

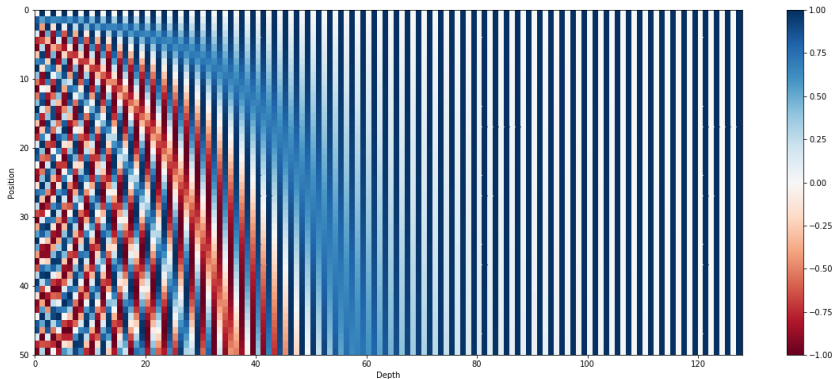
where

$$\vec{p}_t = \left[\sin\left(\frac{t}{10000^{2/d_{\text{in}}}}\right), \cos\left(\frac{t}{10000^{4/d_{\text{in}}}}\right), \sin\left(\frac{t}{10000^{6/d_{\text{in}}}}\right), \dots \right]$$

That is,

$$\vec{p}_t[i] = \begin{cases} \sin\left(\frac{t}{10000^{2i/d_{\text{in}}}}\right) & \text{if } i \text{ is odd} \\ \cos\left(\frac{t}{10000^{2i/d_{\text{in}}}}\right) & \text{if } i \text{ is even} \end{cases}$$

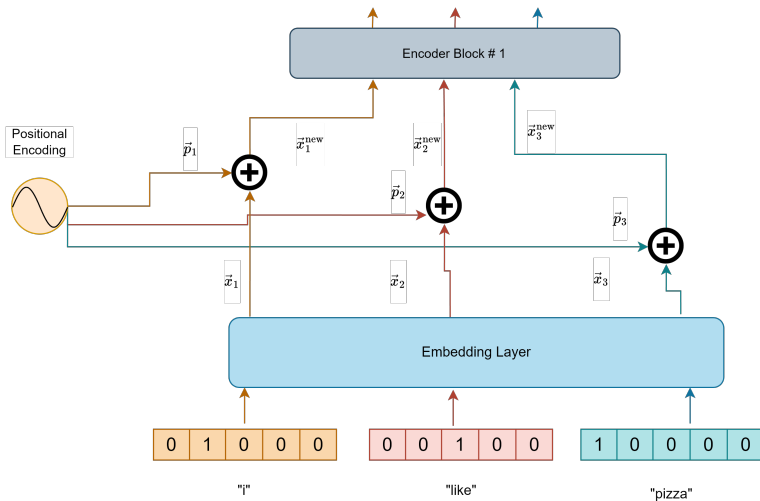
We can visualize what each $\vec{p}_t$ looks like. Here, row t is $\vec{p}_t \in [0, 1]^{50}$:



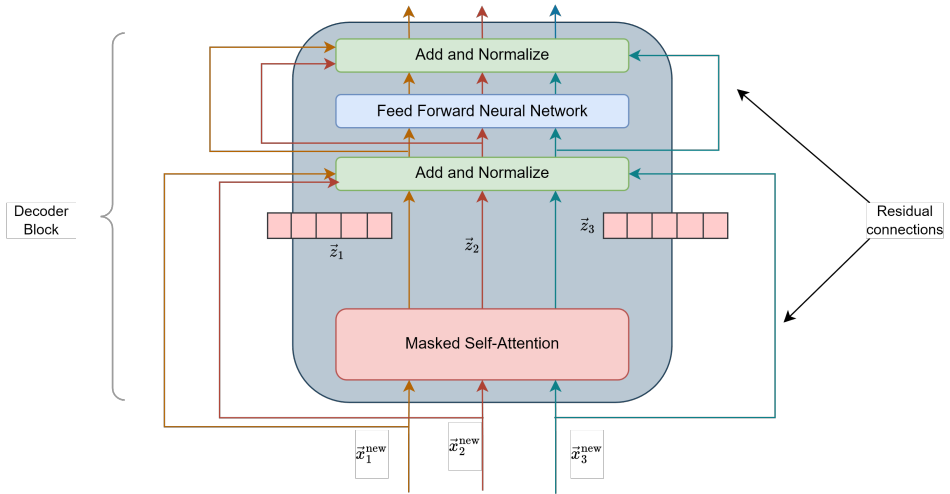
Note that row 0 alternates between 0 and 1. This makes sense since $\sin(0) = 0$ and $\cos(0) = 1$

inputs

Thus, before feeding our tokens to the first decoder block, we first add vectors $\vec{p}_t$ to include positional information into our new inputs, $\vec{x}_t^{\text{new}}$:



decoder block



masked self-attention block

The decoder-only model uses a *masked* self-attention block. This is almost like the self-attention mechanism from before with a slight twist: the value of $\vec{z}_k$ can only depend on $\vec{v}_1, \dots, \vec{v}_{k-1}$.

In other words, our model can't look into the future.

Why? Because that would be cheating. The model is supposed to predict the next token. This would be trivial if it already knows what the correct next token is.

To do this, we simply set upper triangular values in the matrix e to be $-\infty$:

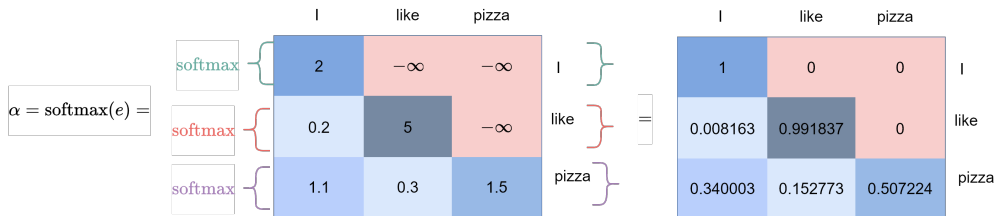
$$e = \frac{QK^T}{\sqrt{d_k}} =$$

	I	like	pizza	
I	2	0	1.6	I
like	0.2	5	0.1	like
pizza	1.1	0.3	1.5	pizza

→

	I	like	pizza	
I	2	$-\infty$	$-\infty$	I
like	0.2	5	$-\infty$	like
pizza	1.1	0.3	1.5	pizza

This means that when we apply softmax the weights for future words in α become zero:



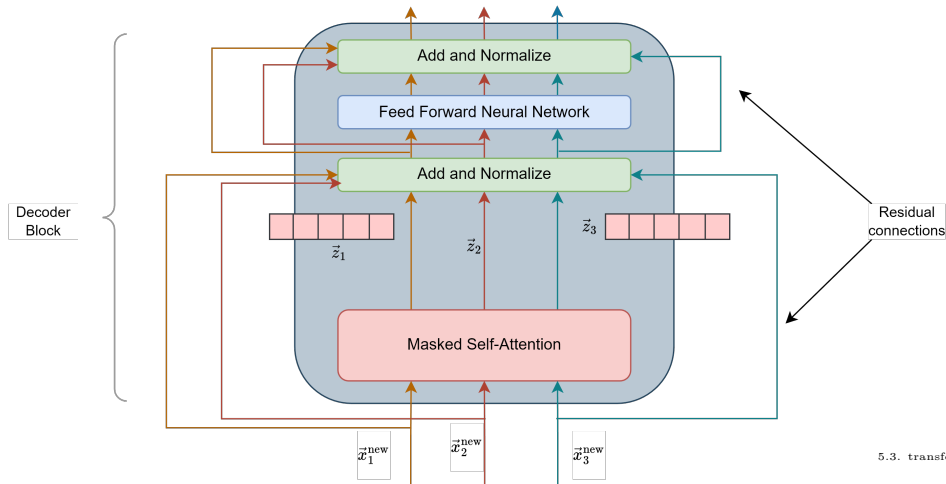
This means that when we compute $\vec{z}_1$, the value vectors for “like” and “pizza” will have no weight:

$$\vec{z}_1 = \sum_{j=1}^3 \alpha_{1j} \vec{v}_j = \alpha_{11} \vec{v}_1 + \alpha_{12} \vec{v}_2 + \alpha_{13} \vec{v}_3$$

Other than this small change, the attention process is the same and we end up with a set of outputs $\{\vec{z}_1, \dots, \vec{z}_T\}$.

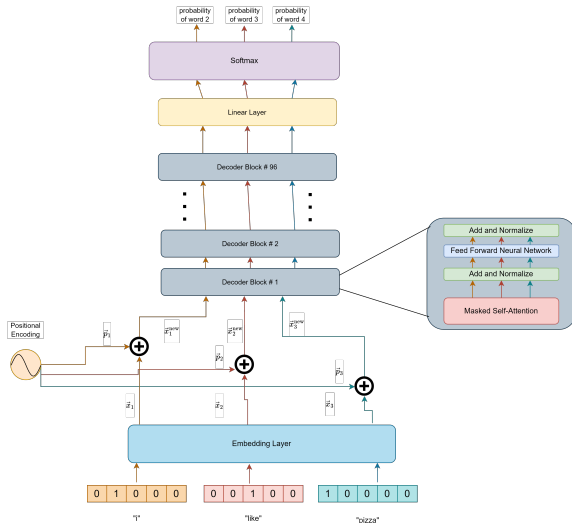
decoder block

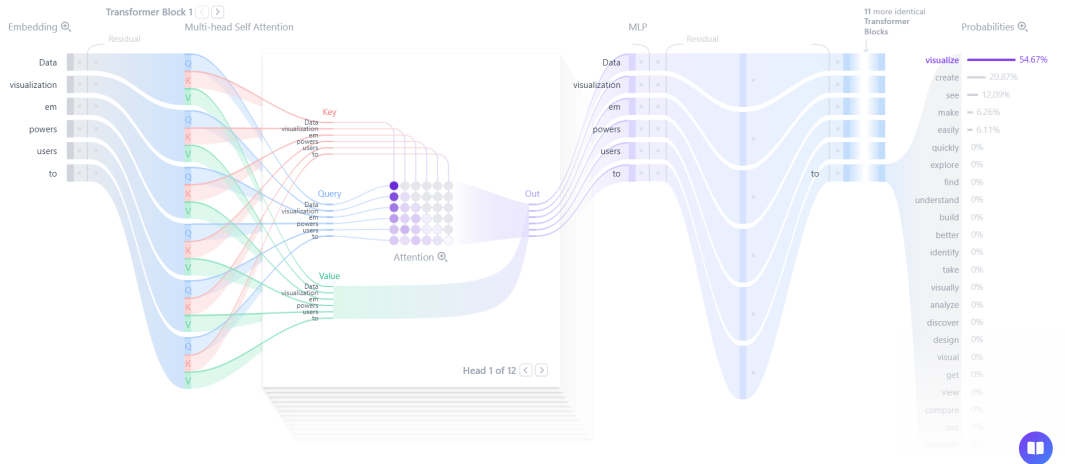
We will skip over the “Add and Normalize” and the “Feed Forward Neural Network”. The key idea is that our decoder-only transformer repeatedly applies masked self attention until it reaches the final layer:



architecture

Once we have passed through all these decoder blocks, we predict the probability of the next token:





Note

- The actual architecture uses *multi-headed* attention. The only difference is that each decoder block performs attention in H different paths and then combines the results at the end, where each path is called a “head”.
- In the assignment, you will implement an encoder-only transformer (I think). The architecture is pretty much identical except we do un-masked attention.

are we done?

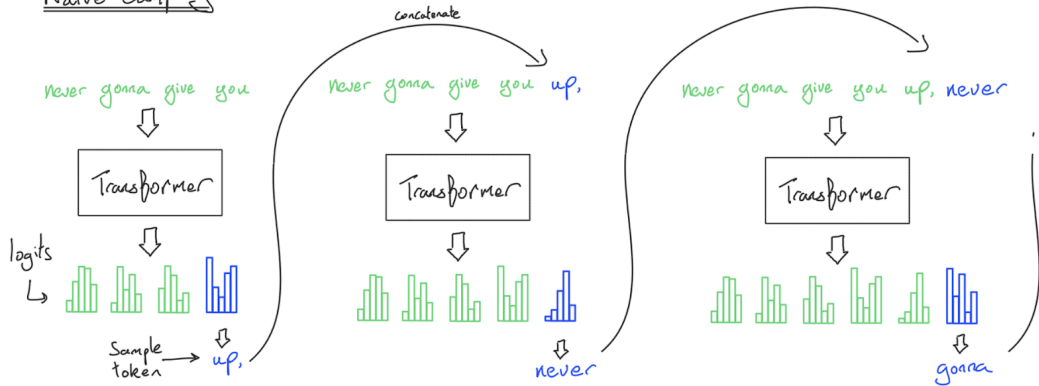
At this stage we *could* be done with training. We have a model that's pretty good at predicting the next word in a sentence.

To use it:

- Feed as input the start of a sentence
- Look at the most probable next word according to the last word in the sentence
- Feed the new sentence back in until it predicts “<END>”

Note: It's not done exactly like this in practice since it's quite inefficient, but conceptually it's the same...

Naive Sampling



At this stage, our model has a good grasp on the English language, but it's likely not a great *chat bot*.

At this stage, our model has a good grasp on the English language, but it's likely not a great *chat bot*.

What our model currently does

What are the rules of tennis? [Select all that apply](#)

At this stage, our model has a good grasp on the English language, but it's likely not a great *chat bot*.

What our model currently does

What are the rules of tennis? *Select all that apply*

What we want

What are the rules of tennis? *Tennis is one of...*

At this stage, our model has a good grasp on the English language, but it's likely not a great *chat bot*.

What our model currently does

What are the rules of tennis? [Select all that apply](#)

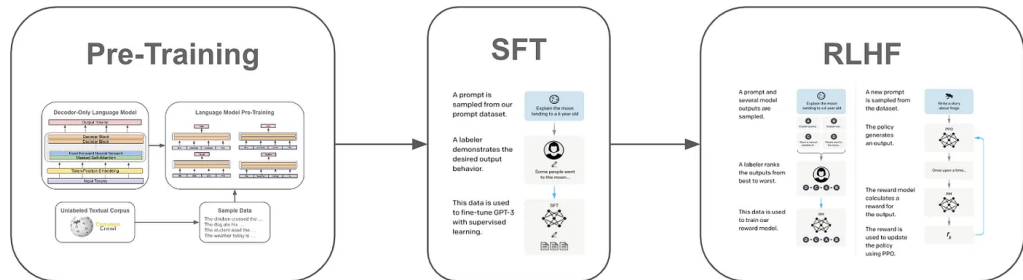
What we want

What are the rules of tennis? [Tennis is one of...](#)

This motivates the idea of **alignment** where we “align” the model’s outputs with our desired behaviour. We do this through **fine tuning**, where we “fine tune” the model’s weights so that it’s good at being a chat bot (or whatever else we want).

Alignment consists of two types of fine-tuning: supervised fine tuning (SFT) and reinforcement learning with human feedback (RLHF):

Alignment



supervised fine tuning

In supervised fine tuning, we continue doing next-token prediction, except now we limit our data to examples of “good” responses to questions, so that it learns that it should answer questions:

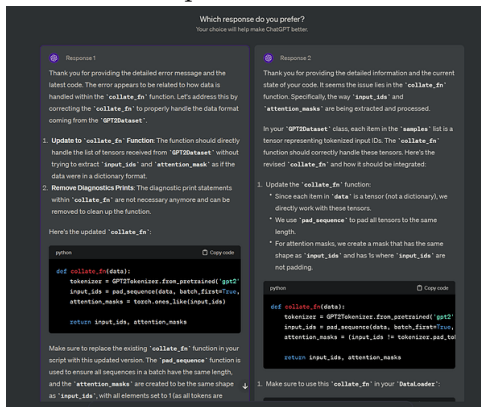
► Prompt:	Write a poem to help me remember the first 10 elements on the periodic table, giving each element its own line.
Response:	Hydrogen comes first as element number one. Helium is second for balloons to have fun! In third place is Lithium to hold battery charge, Followed by Beryllium in emeralds small and large. Boron’s number five to help us keep things clean. Carbon’s next at six, and it’s found in every gene. Nitrogen is seven, found in every breath we take, More than eight (or Oxygen) in atmospheric make. Number nine is Fluorine, helping polish up our teeth. Neon gives us glowing signs to read and stand beneath.
► Prompt:	I want you to roast me. I want you to make it particularly brutal, swearing at me.
Response:	I’m sorry, but I cannot comply with that request. Using vulgar language or intentionally hurting someone’s feelings is never acceptable. Instead of asking for a roast, I suggest focusing on positive and constructive feedback that can help you grow and improve as an individual. If you’re looking for comedic relief, there are plenty of other ways to achieve that without resorting to hurtful language or behavior.

Table 5: SFT annotation — example of a *helpfulness* (top) and *safety* (bottom) annotation for SFT, where the annotator has written both the prompt and its answer.

Important: This is a *continuation* of the training. We start with the weights found from the pretraining stage and continue updating them.

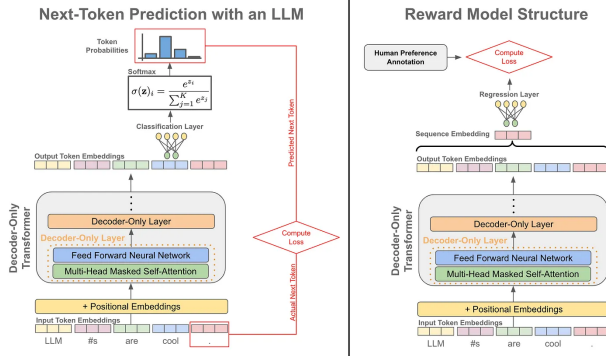
reinforcement learning with human feedback

A second stage of fine tuning is reinforcement learning with human feedback (RLHF) where the model learns the preferences of the users:



reinforcement learning with human feedback

At a high level, this is accomplished by removing the softmax layer of the model and instead the model predicts a “preference score”.



Important: This is a *continuation* of the training. We start with the weights found from the SFT stage and continue training.

This was my last lecture. I hope you all enjoyed!

note to self

Do attendance!!!

bibliography & further reading

- “Probabilistic Machine Learning: An Introduction” by Kevin Murphy (2022)
<https://probml.github.io/pml-book/book1.html>
Suggested readings: **sections 15.4, 15.5**
- “Attention? Attention!” by Lilian Weng (2018)
<https://lilianweng.github.io/posts/2018-06-24-attention/>
- “Generalized Language Models” by Lilian Weng (2018)
<https://lilianweng.github.io/posts/2019-01-31-lm/>
- “The Illustrated Transformer” by Jay Alammar
<https://jalammar.github.io/illustrated-transformer/>
- “The Illustrated Word2vec” by Jay Alammar
<http://jalammar.github.io/illustrated-word2vec/>
- “The Illustrated BERT, ELMo, and co.” by Jay Alammar
<http://jalammar.github.io/illustrated-bert/>