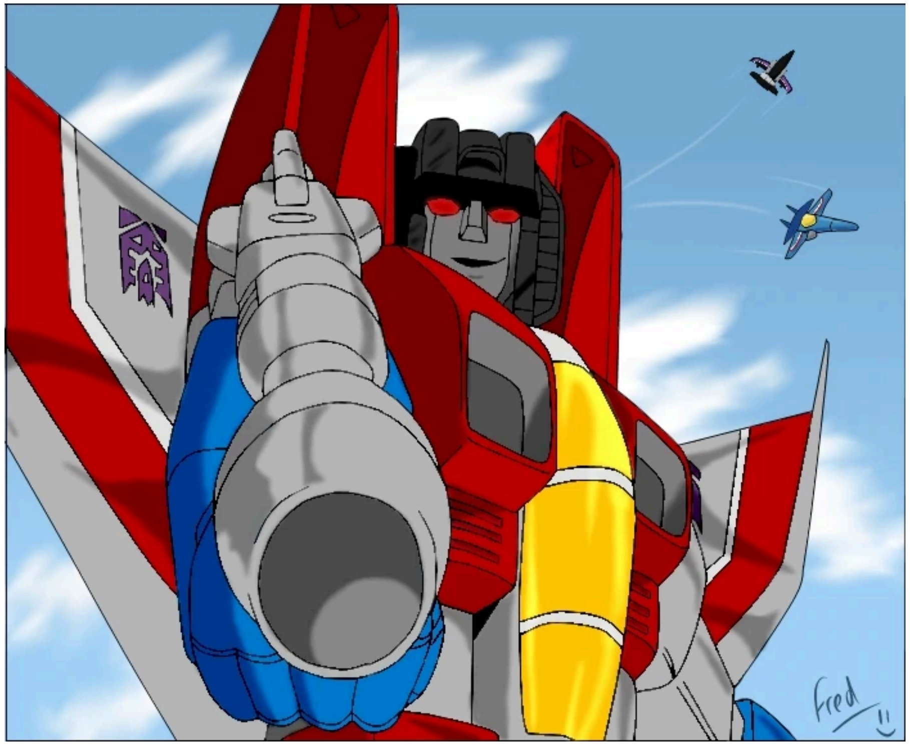
Large language models and deep learning for computer vision and time series: From academia to industry

Transformers!!! (more cool) things to learn and to do



Felipe Ferreira de Freitas

felipe.defreitas@edfenergy.com

my github

But first, how did we get here?

Of course, to tell the whole history of machine learning and natural language processing (NLP) would take quite some time. However, we can point towards some pivotal points in the field, starting from the work of Michael I. Jordan in 1986 Serial Order: A Parallel Distributed Processing Approach , where it was trained a tiny the network with only a handful of neurons to predict simple sequences of two symbols 1010101001 or ABABBAB

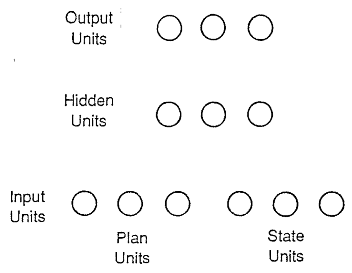


FIGURE 2. The processing units in the network. The plan and state units together constitute the input units for the network.

Michael I. Jordan

10 MICHAEL I. JORDAN

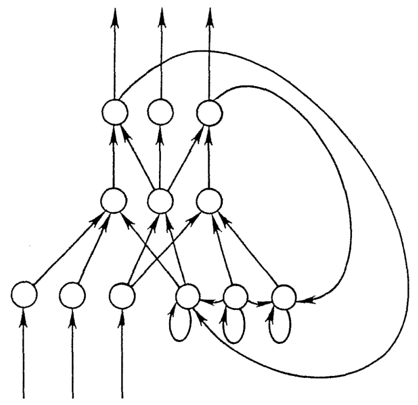


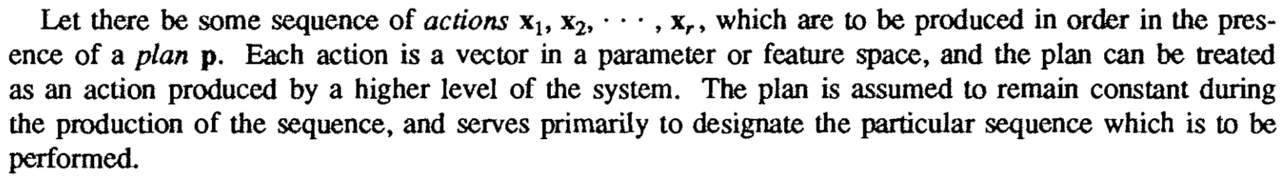
FIGURE 3. Basic network connection scheme (not all connections are shown).

Michael I. Jordan

The key idea was to **"mimic" how we "think" our mind works** - where we have an ongoing "state of mind" in which we use to decide next actions to be taken. This was similar to give memory to the neural networks.

how did we get here?

Another good idea was how the "training" method was defined:



Let there be some sequence of *actions* $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_r$, which are to be produced in order in the presence of a *plan* $\mathbf{p}$. Each action is a vector in a parameter or feature space, and the plan can be treated as an action produced by a higher level of the system. The plan is assumed to remain constant during the production of the sequence, and serves primarily to designate the particular sequence which is to be performed.

where the learning signal (loss) is the difference between the network's guess of the next symbol and the real data. And not only that, after training the NN Michael I. Jordan, set up experiments in which the network would produce new sequences using the outputs from previous sequences.

The NN would initially generate some mistakes, but such mistakes would go away after the NN was trained for a bit more time.

The RNN was also trained in spatial data patterns:

The learned sequences where not only memorized, they were generalized.

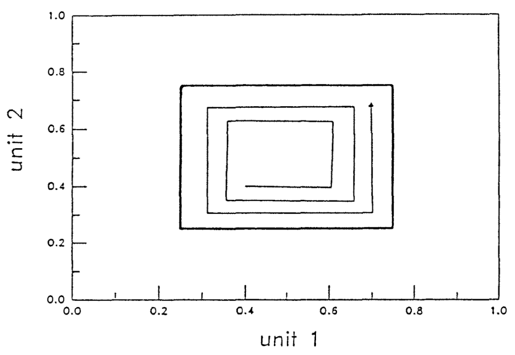


FIGURE 8. The activations of the two output units plotted with time as a parameter. The square is the trajectory that the network learned, and the spiral trajectory is the path that the network followed when started at the point (.4, .4).

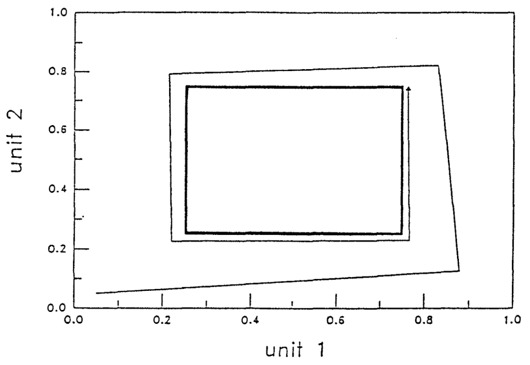


FIGURE 9. The activations of the two output units plotted with time as a parameter. The square is the trajectory that the network learned, and the spiral trajectory is the path that the network followed when started at the point (.05, .05).

Michael I. Jordan

Michael I. Jordan

in his words -> "An important fact about the learned trajectories is that they tend to influence points nearby in the state space. Indeed, the learned trajectories tend to be **attractors**."

5 years latter

Jeffrey Elman, picked up on Jordan’s research and did this same thing with a slightly bigger network (~50 neurons) and trained the network on language:

- "The sequence was composed of six different 6-bit binary vectors. Although the vectors were not derived from real speech, one might think of them as representing speech sounds, with the six dimensions of the vector corresponding to articulatory features."

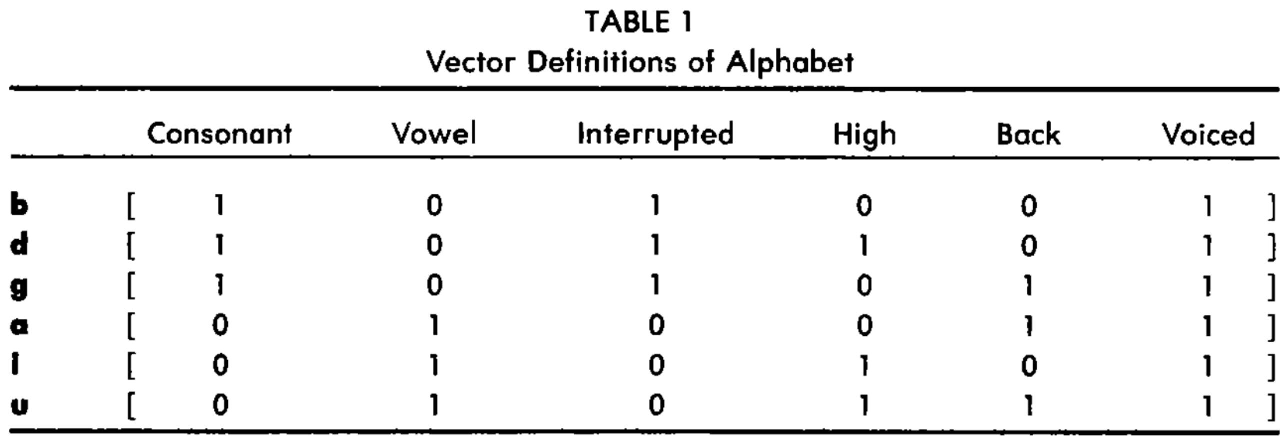


TABLE 1
Vector Definitions of Alphabet

	Consonant	Vowel	Interrupted	High	Back	Voiced
b	[1	0	1	0	0	1]
d	[1	0	1	1	0	1]
g	[1	0	1	0	1	1]
a	[0	1	0	0	1	1]
i	[0	1	0	1	0	1]
u	[0	1	0	1	1	1]

- The sequence was semi-random; consonants occurred randomly, but following a given consonant, the identity and number of following vowels was regular

FINDING STRUCTURE IN TIME

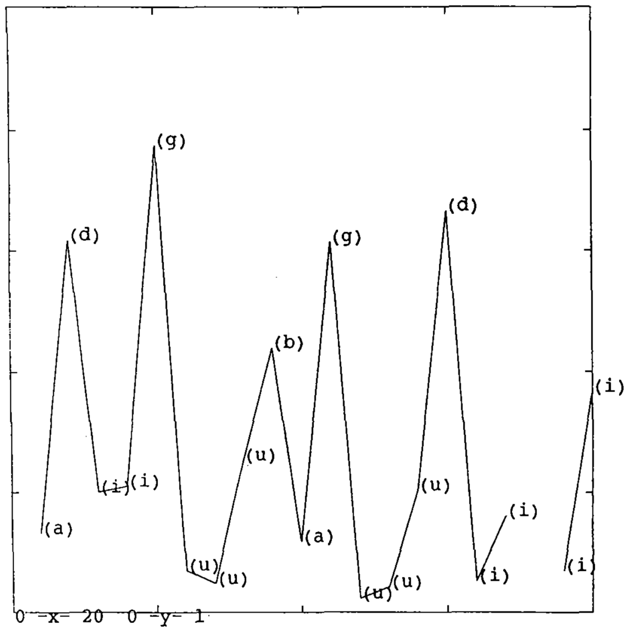


Figure 4. Graph of root mean squared error in letter prediction task. Labels indicate the correct output prediction at each point in time. Error is computed over the entire output vector.

Jeffrey Elman

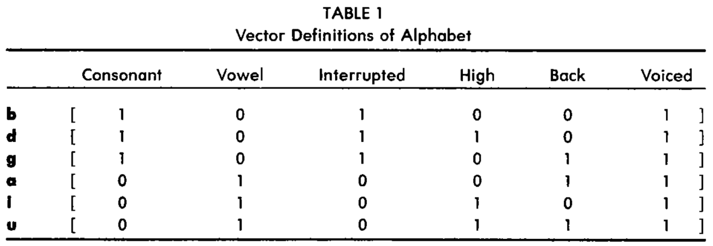


TABLE 1
Vector Definitions of Alphabet

		Consonant	Vowel	Interrupted	High	Back	Voiced	
b	[	1	0	1	0	0	1	]
d	[	1	0	1	1	0	1	]
g	[	1	0	1	0	1	1	]
a	[	0	1	0	0	1	1	]
i	[	0	1	0	1	0	1	]
u	[	0	1	0	1	1	1	]

Jeffrey Elman

Elman noted that Error on predicting the first bit was consistently lower than error for the fourth bit, but why?

- The first bit represents the Consonant feature.
- The fourth bit corresponds to the High feature.
- While all consonants share the same value for the Consonant feature, they vary in terms of the High feature.
- The network has acquired knowledge about the sequential relationship between consonants and vowels, leading to low error rates on vowels.
- Additionally, the network has learned the frequency of vowels following each consonant.

DISCOVERING THE NOTION “WORD”

As Elman pointed out $\rightarrow$ "... more extended sequential dependencies may not necessarily be more difficult to learn. If the dependencies are structured, that structure may make learning easier and not harder."

So, he designed a similar task, but now the sequences would form a real words. The letters are presented in sequence, one at a time, with no breaks between the letters in a word, and no breaks between the words of different sentences.

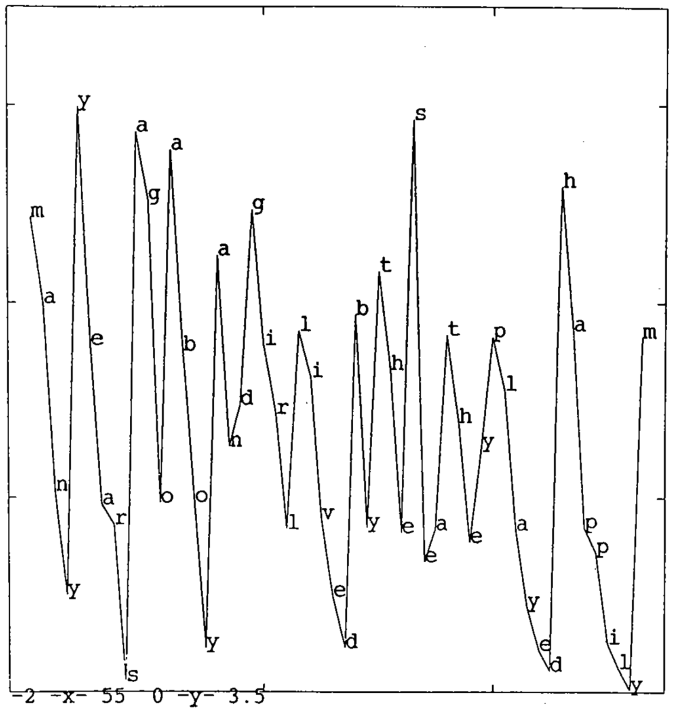


Figure 6. Graph of root mean squared error in letter-in-word precition task.

Jeffrey Elman

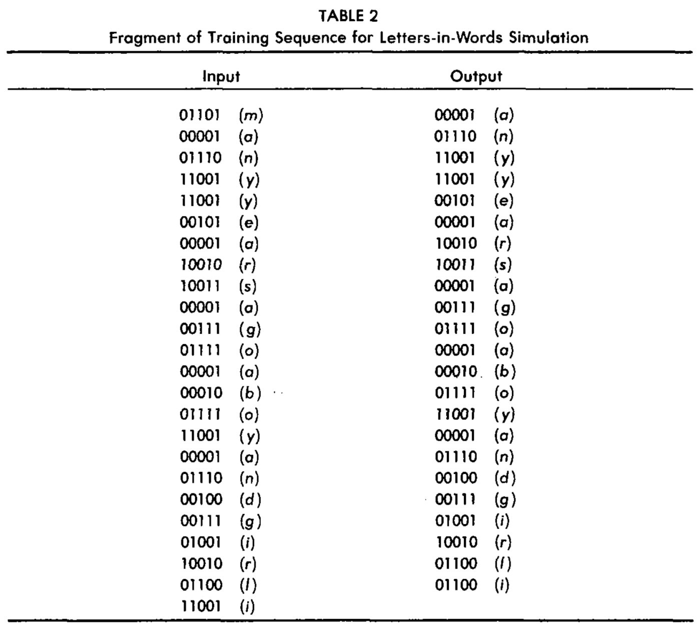


TABLE 2
Fragment of Training Sequence for Letters-in-Words Simulation

Input	Output
01101 (*m*)	00001 (*a*)
00001 (*a*)	01110 (*n*)
01110 (*n*)	11001 (*y*)
11001 (*y*)	11001 (*y*)
11001 (*y*)	00101 (*e*)
00101 (*e*)	00001 (*a*)
00001 (*a*)	10010 (*r*)
10010 (*r*)	10011 (*s*)
10011 (*s*)	00001 (*a*)
00001 (*a*)	00111 (*g*)
00111 (*g*)	01111 (*o*)
01111 (*o*)	00001 (*a*)
00001 (*a*)	00010 (*b*)
00010 (*b*)	01111 (*o*)
01111 (*o*)	11001 (*y*)
11001 (*y*)	00001 (*a*)
00001 (*a*)	01110 (*n*)
01110 (*n*)	00100 (*d*)
00100 (*d*)	00111 (*g*)
00111 (*g*)	01001 (*i*)
01001 (*i*)	10010 (*r*)
10010 (*r*)	01100 (*l*)
01100 (*l*)	01100 (*i*)
11001 (*i*)	

Jeffrey Elman

"at the onset of a new word the chance of error is high, as more of the word is received the error rate declines, since the sequence is increasingly predictable…" at the end of a word the error jumps up again, following the same pattern.

DISCOVERING LEXICAL CLASSES FROM WORD ORDER

Elman continued his experiments and proposed an interesting question of whether a network can learn any aspects of that underlying abstract structure, like if the network could learn semantic structures encoded in language?

To answer this question, he implemented a new bigger network with 362 neurons (150 of them acting as context units where learned patterns could be stored). And he created more data, 10,000 two- & three-word sentences.

To understand what his network was doing he probed the internal neurons in the context unit as it was processing words. (which took the form of 150 bit vectors), and then he plotted them and compared the spatial arrangement.

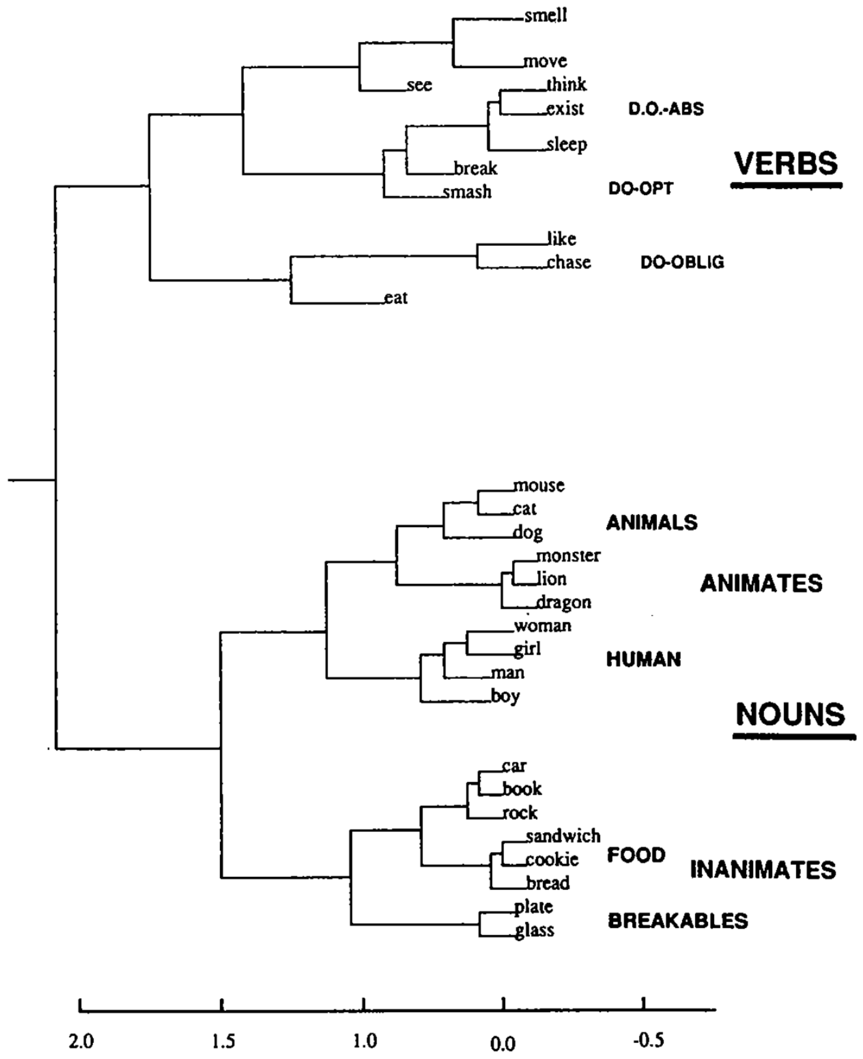


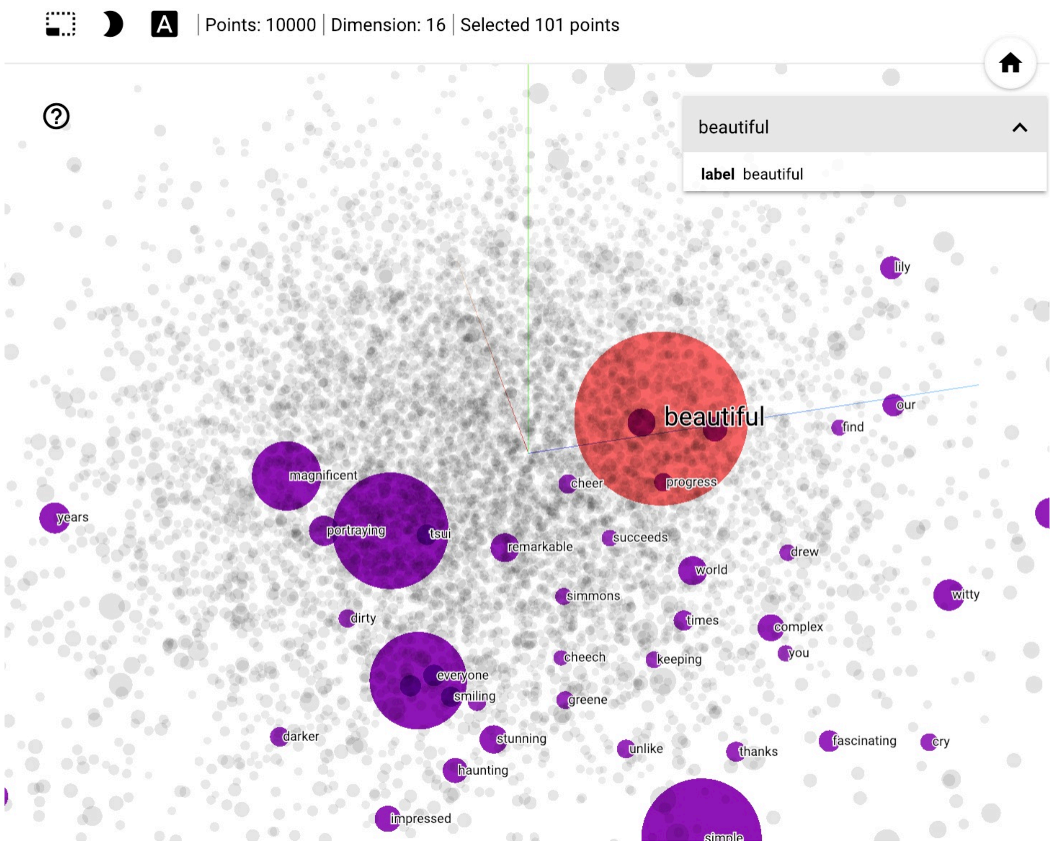
Figure 7. Hierarchical cluster diagram of hidden unit activation vectors in simple sentence prediction task. Labels indicate the inputs which produced the hidden unit vectors; inputs were presented in context, and the hidden unit vectors averaged across multiple contexts.

Jeffrey Elman

The network would spatially cluster words based on meaning, for example it separated nouns which are inanimate (car, rock) and animate (girl, boy), and within these groups he saw subcategorization, for example the animate objects were broken down into "human" and "non-human" clusters, and the "non-human" cluster broke down into "large animals" and "small animals"...inanimate objects were broken down into "breakable" and "edible"...

DISCOVERING LEXICAL CLASSES FROM WORD ORDER

Another insight Elman had was the representation of words as vectors in high dimensional space, then sequences of words could be thought of as a pathway in this space, and similar words would have similar pathways



Word embeddings

Following paths and pay attention

It wasn’t until 2011, when Ilya Sutskever, James Martens and Geoffrey Hinton pushed the idea of word vectors and high dimensional pathways ahead on a much larger model and made a bold observation in terms of the connection between prediction and intelligence.

In the paper Generating Text with Recurrent Neural Networks they demonstrate the use of large RNNs (trained with the new Hessian-Free optimizer) in the task of predicting the next character in a stream of text.

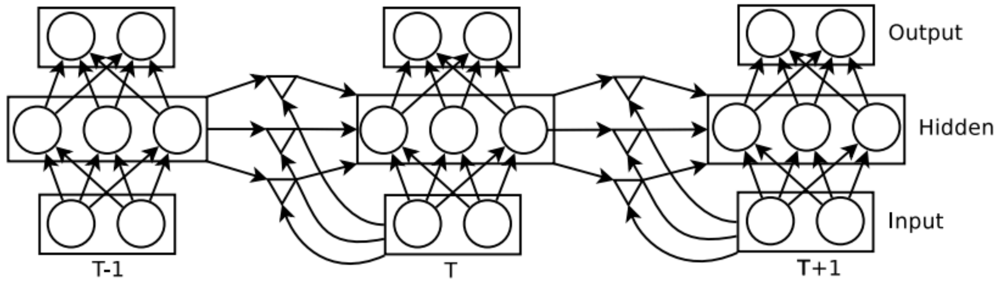
An interesting application for the solution to this task was a better improve in the compression of text files, and in their words $\rightarrow$ "More speculatively, achieving the asymptotic limit in text compression requires an understanding that is “equivalent to intelligence” (Hutter, 2006)."

This was in line with one theory that biological brains, at their core, are prediction machines.

- If intelligence is the ability to learn.
- This views learning as the compression of experience into a predictive model of the world.

Following paths and pay attention

They trained a much larger RNN, which they refereed as Multiplicative (or "gated") Recurrent Neural Network. This new “MRNN” architecture uses multiplicative connections to allow the current input character to determine the hidden-to-hidden weight matrix.



Ilya Sutskever, James Martens and Geoffrey Hinton

"We trained MRNNs on over a hundred of megabytes of text for several days using 8 Graphics Processing Units in parallel to perform significantly better than one of the best word-agnostic single character-level language models: the sequence memoizer" (very likely training in a gt9800 series)

To generate phrases they first feed in the phrase "The meaning of life is", to which it continues:

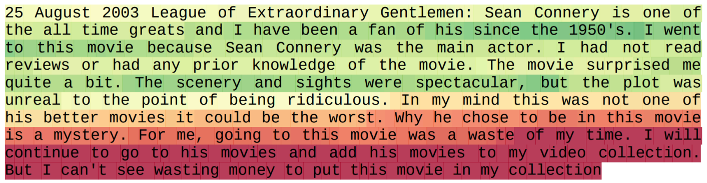
***The meaning of life is the tradition of the ancient human reproduction:** it is less favorable to the good boy for when to remove her bigger. In the show's agreement unanimously resurfaced. The wild pasteured with consistent street forests were incorporated by the 15th century BE. In 1996 the primary rapford undergoes an effort that the reserve conditioning, written into Jewish cities, sleepers to incorporate the .St Eurasia that activates the population. Mar??a Nationale, Kelli, Zedlat-Dukastoe, Florendon, Ptu's thought is. To adapt in most parts of North America, the dynamic fairy Dan please believes, the free speech are much related to the*

To their surprise the trained MRNN exhibited a significant amount of interesting and high-level linguistic structure. However, beyond few words it would quickly fall into a "non-sense" pathway. It was clear that some sort of learning was happening, but they were hitting the capacity of the network to maintain coherent context over long sequences.

Following paths and pay attention:

Uncover sentiments from words

In 2017 when a team of researchers (Alec Radford, Rafal Jozefowicz, **Ilya Sutskever**, at a speculative lab called Open AI), and setup a larger recurrent network (an LSTM) and trained it on a massive set of 82 million amazon and IMDB reviews - this training process took a month. The largest model to date.



25 August 2003 League of Extraordinary Gentlemen: Sean Connery is one of the all time greats and I have been a fan of his since the 1950's. I went to this movie because Sean Connery was the main actor. I had not read reviews or had any prior knowledge of the movie. The movie surprised me quite a bit. The scenery and sights were spectacular, but the plot was unreal to the point of being ridiculous. In my mind this was not one of his better movies it could be the worst. Why he chose to be in this movie is a mystery. For me, going to this movie was a waste of my time. I will continue to go to his movies and add his movies to my video collection. But I can't see wasting money to put this movie in my collection

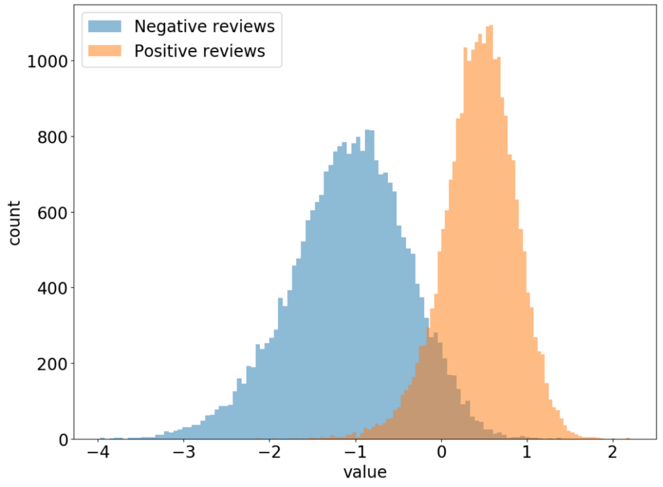


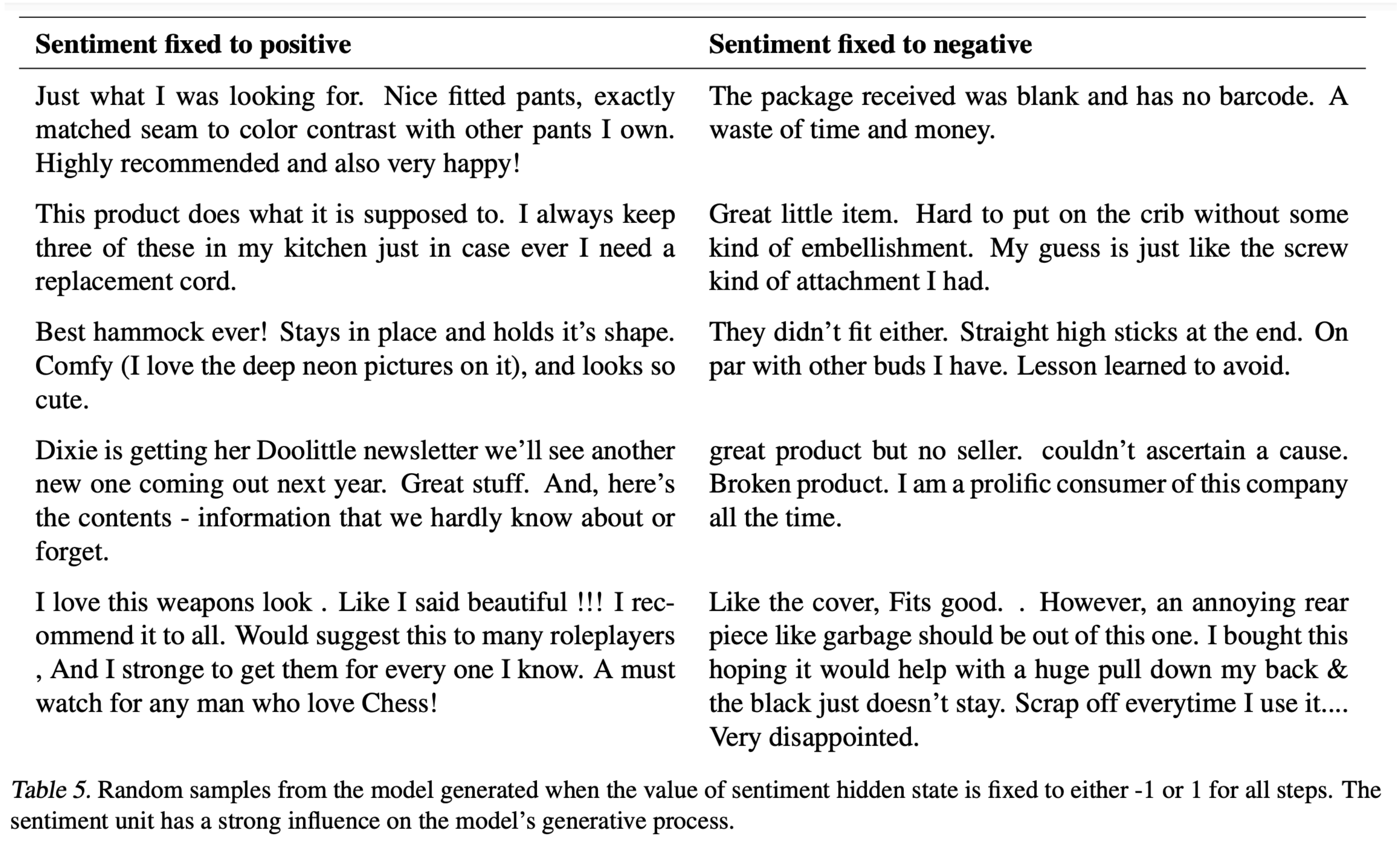
Figure 3. Histogram of cell activation values for the sentiment unit on IMDB reviews.

"...we discovered a single unit within the mLSTM that directly corresponds to sentiment."

Following paths and pay attention

Uncover sentiments from words

To show this network had a good internal model (or understanding) of sentiment, they had the network generate text, and while doing so they forced the sentiment neuron to be a positive vs negative value, and it spit out positive and negative reviews which were entirely artificial, but indistinguishable from a human review.



Sentiment fixed to positive	Sentiment fixed to negative
Just what I was looking for. Nice fitted pants, exactly matched seam to color contrast with other pants I own. Highly recommended and also very happy!	The package received was blank and has no barcode. A waste of time and money.
This product does what it is supposed to. I always keep three of these in my kitchen just in case ever I need a replacement cord.	Great little item. Hard to put on the crib without some kind of embellishment. My guess is just like the screw kind of attachment I had.
Best hammock ever! Stays in place and holds it's shape. Comfy (I love the deep neon pictures on it), and looks so cute.	They didn't fit either. Straight high sticks at the end. On par with other buds I have. Lesson learned to avoid.
Dixie is getting her Doolittle newsletter we'll see another new one coming out next year. Great stuff. And, here's the contents - information that we hardly know about or forget.	great product but no seller. couldn't ascertain a cause. Broken product. I am a prolific consumer of this company all the time.
I love this weapons look . Like I said beautiful !!! I recommend it to all. Would suggest this to many roleplayers , And I stronge to get them for every one I know. A must watch for any man who love Chess!	Like the cover, Fits good. . However, an annoying rear piece like garbage should be out of this one. I bought this hoping it would help with a huge pull down my back & the black just doesn't stay. Scrap off everytime I use it.... Very disappointed.

Table 5. Random samples from the model generated when the value of sentiment hidden state is fixed to either -1 or 1 for all steps. The sentiment unit has a strong influence on the model's generative process.

But simply going bigger with more data was hitting a practical limit at the time

there was still a key problem with RNN's, which is that they processed data serially, one word at a time. And all the contexts had to be squeezed into the fixed internal memory, i.e. the RNNs were struggling to connect concepts the longer the context was.

Following paths and pay attention

An alternative to RNN's tried to tackle this problem by simply processing the entire input sequence of text in parallel by passing it through a very wide, fully connected, network -> The Sparsely-Gated Mixture-of-experts layer.

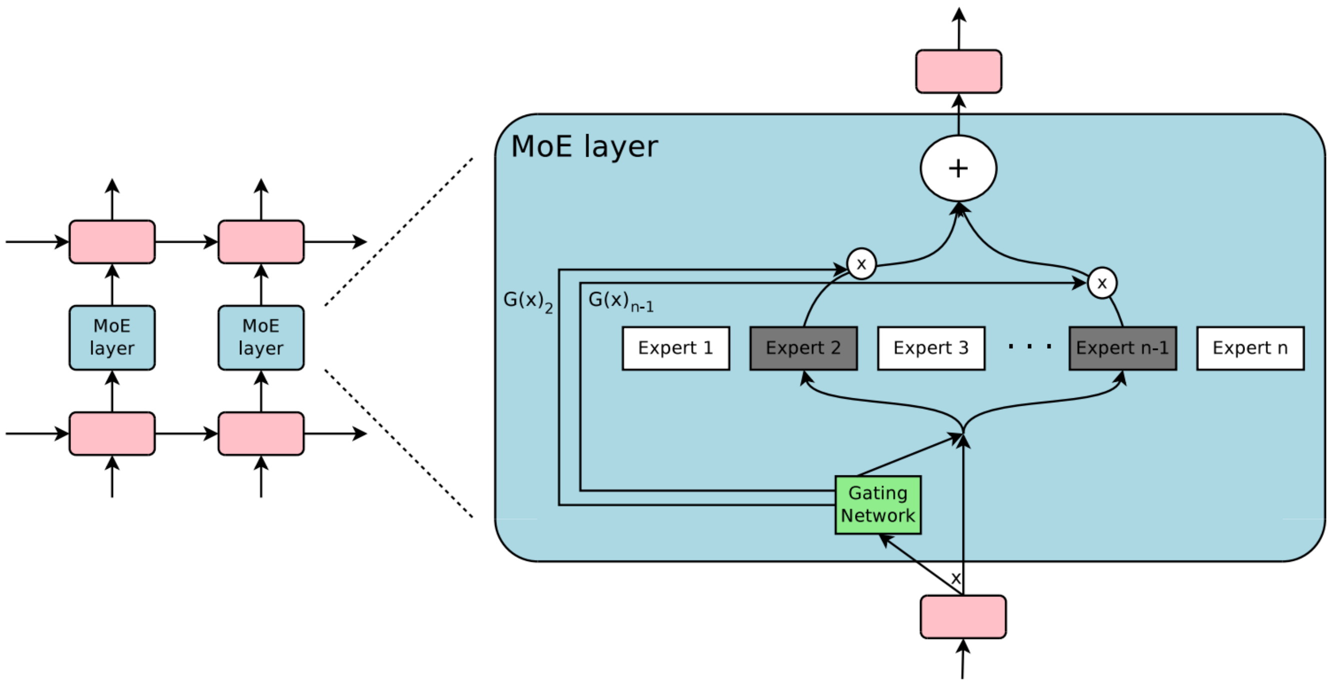


Figure 1: A Mixture of Experts (MoE) layer embedded within a recurrent language model. In this case, the sparse gating function selects two experts to perform computations. Their outputs are modulated by the outputs of the gating network.

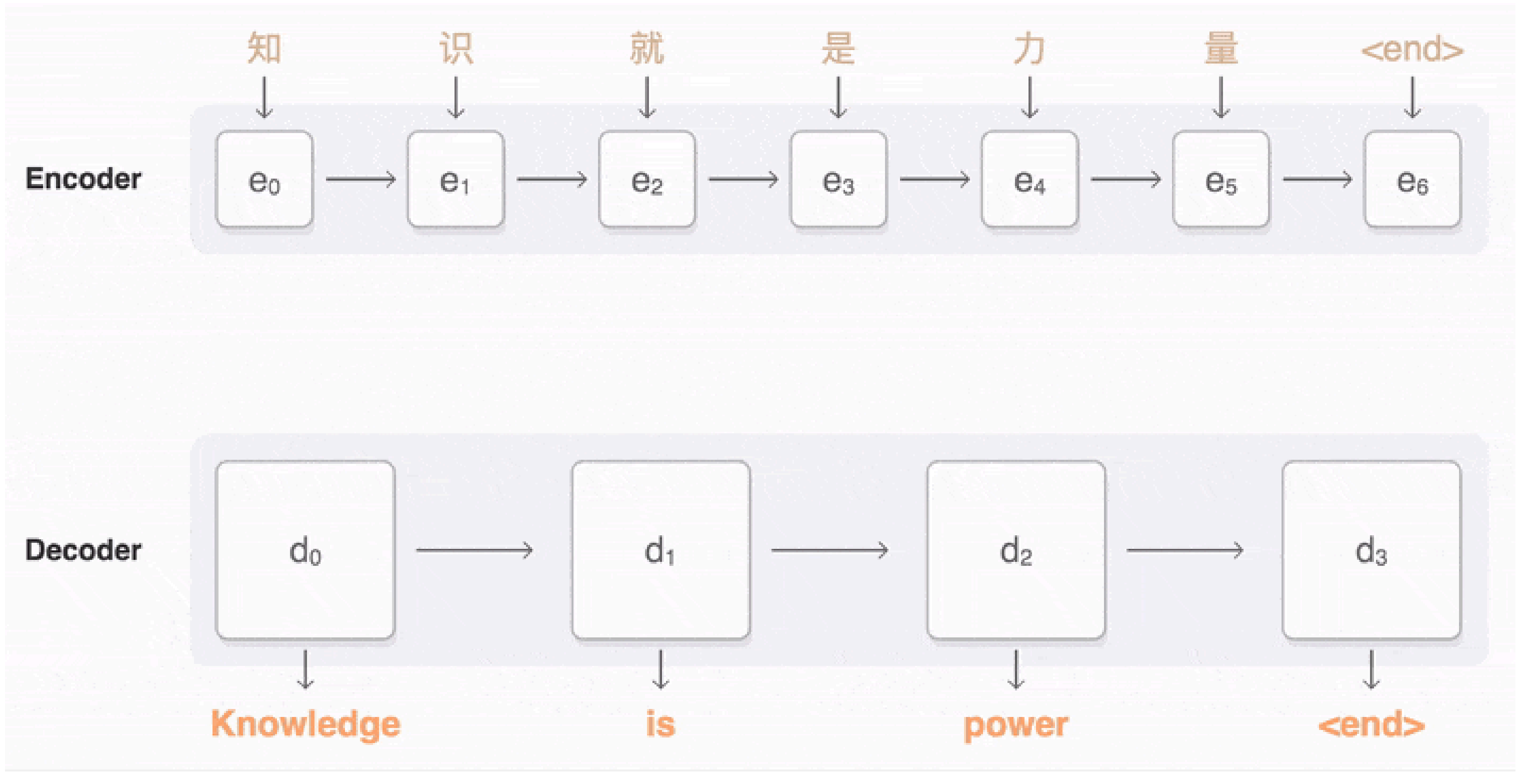
Noam Shazeer, et. al.

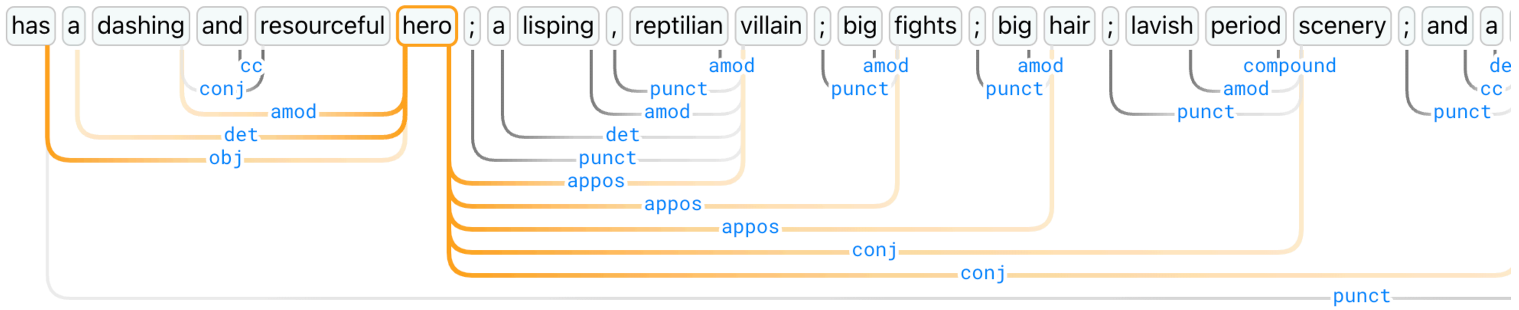
But this was impractical, since it requires many layers of depth to compensate the lack memory, since it builds up the context of the sentence across multiple layers. This approach is tempting but the resulting network becomes impossible to train.

The rise of transfromers

In 2017 a ground-breaking paper focused on the problem of translating between languages, one of the key ideas on this paper was a solution to the memory constraint.

The self-attention mechanism:




has
a
dashing
and
resourceful
hero
;
a
lisping
,
reptilian
villain
;
big
fights
;
big
hair
;
lavish
period
scenery
;
and
a
cc
conj
amod
det
obj
amod
punct
amod
det
punct
appos
amod
punct
appos
amod
punct
appos
compound
amod
punct
conj
cc
punct
conj
punct

seq2seq

The rise of transfromers

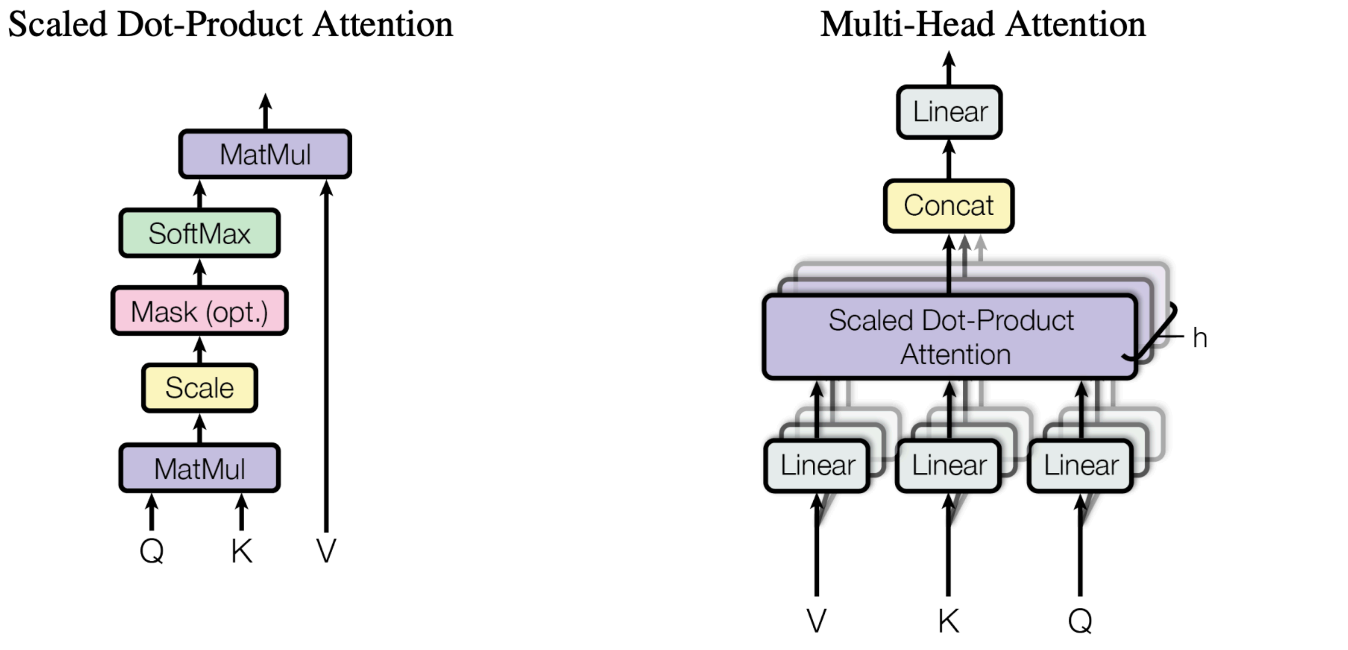


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

These self-attention dynamical layers work by allowing every word in the input to look at & compare itself to every other word, and "absorb the meaning" from the most relevant words to better capture the context of its intended use in that sentence, this is done by a set of smaller neural networks refereed as attention heads.

The rise of transfromers

Self-Attention Process

Take as example the followng phrases:

- I put on a **light** jacket.
- [50256, 40, 1234, 319, 257, **1657**, 15224, 13]
- The room was filled with **light**.
- [50256, 464, 2119, 373, 5901, 351, **1657**, 13]
- She had a **light** workload that day.
- [3347, 550, 257, **1657**, 26211, 326, 1110, 13]

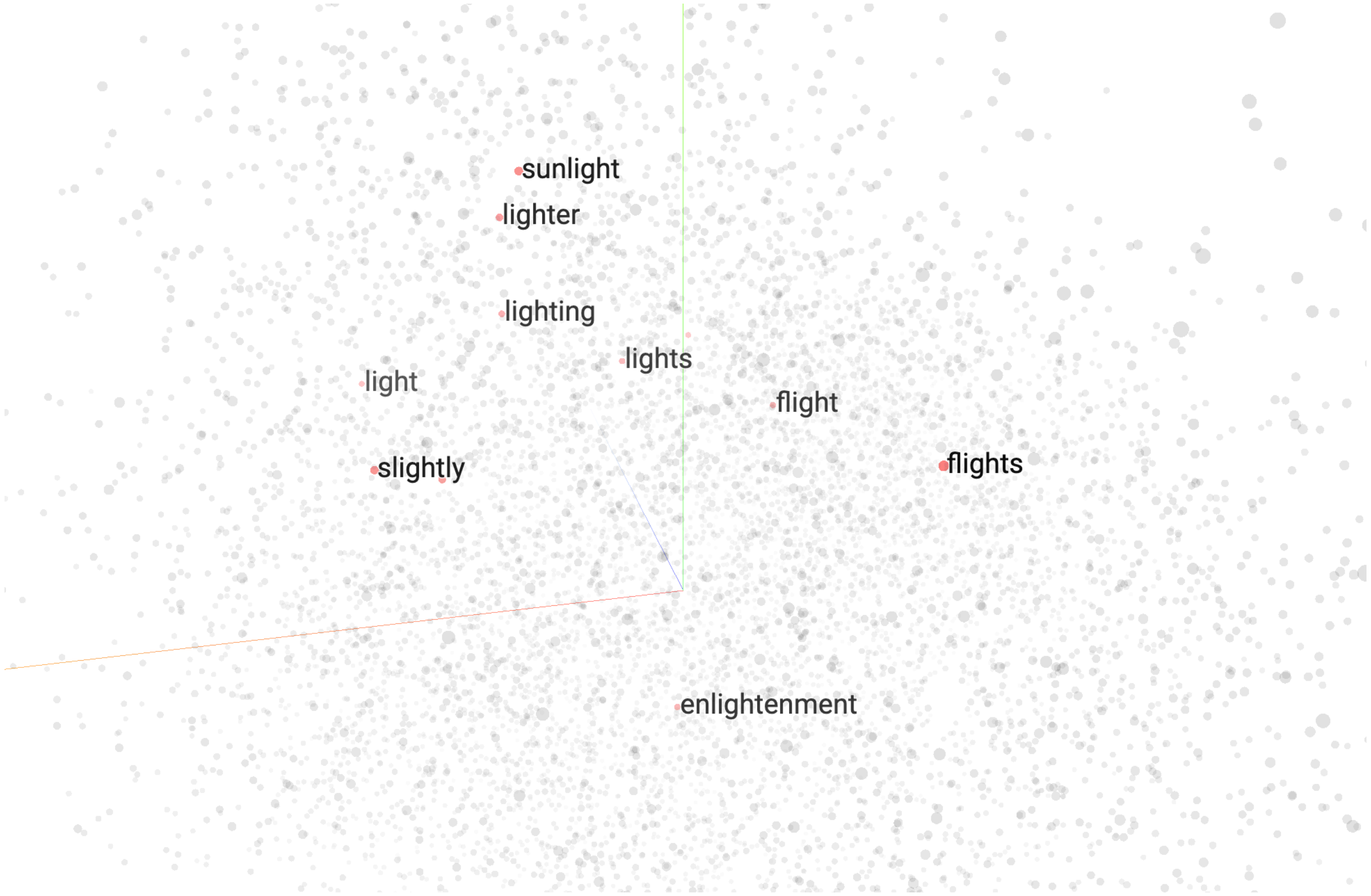
We see that the value for the word light in the vector is the same in all phrases, this initial token set is just a lookup table. It is only in the next step that the transformers allow the surround information to be passed around each embedding vector.

obs: The embedding matrix for a gpt-2 model has 50257x768 (vocab x embedding size), it also have a positional encoding matirx of 1024 x 768, which is a trainable parameter.

The rise of transfromers

Self-Attention Process

In our case the word **light** can be seen as different vectors in the "concept" space, where the directions of these vectors points to the meanings of the word **light** in each context applied.



The main gist of the attention mechanism is to change the embeddings vectors to move in the "concept" space of each word and adjust the direction of each vector (i.e. its meaning) to be more aligned to the context as a whole, hence the name **transformers**.

Self-Attention Process

As initial input we have the embeddings of each token from the phrase "I put on a light jacket.", these embeddings only encoding the meaning of each token and their position and nothing else.

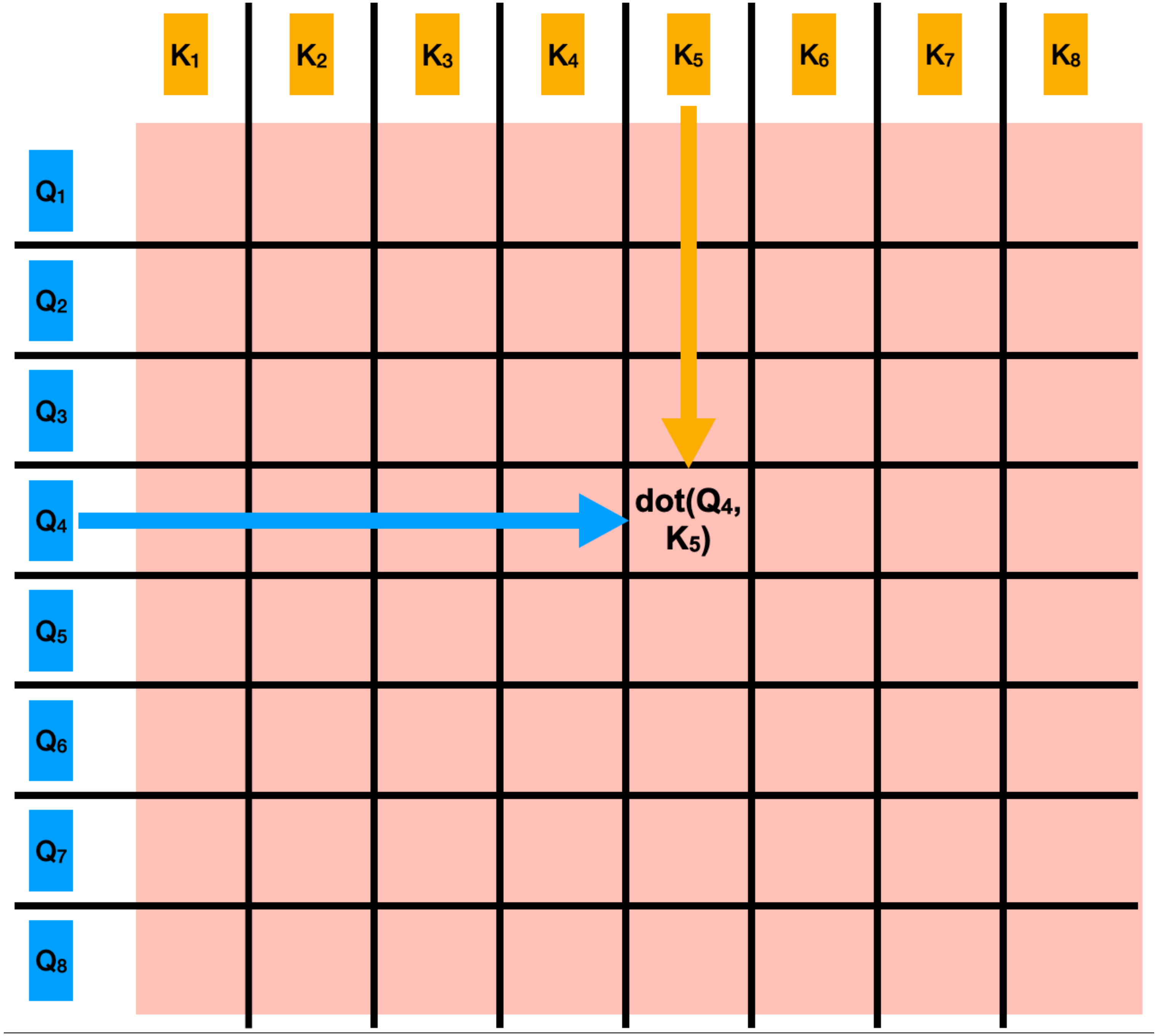
the goal is to have a new refine set of embedding vectors which hopefully will ingest the meaning of the words around in a more broad context and continue the text without fell into non-sense.

1. each tokens "emits" a **Q**uerry, **K**ey and **V**alue, these can be seen as hidden representations of each token, and to allow the mechanism to model relevance, we let each word ask questions and receive answers.

To make it easy to visualize it like we are trying to model the relationship between nouns and adjectives, and in this case each noun is asking to all other words if there is an adjective in front of itself? and this question is encoded, by the matrix W_q, as our **Q**uerry.

At the same time we also have another set of **K**eys, vectors encoded by the matrix W_k, to which we can interpret as the answers to the querries **Q**.

When the question asked by a word matches the answer sent by another word, we will interpret this as the two words having high relevance for each other, in machine learning lingo is said that the embeddings of a given token "attend" to a respective one.


K_1
K_2
K_3
K_4
K_5
K_6
K_7
K_8
Q_1
Q_2
Q_3
Q_4
Q_5
Q_6
Q_7
Q_8
dot(Q_4, K_5)

check here

Self-Attention Process

2. Computing the Attention Matrix: We take the dot product of all the N query vectors with all the N key vectors. Since the dot product of two vectors (no matter how large) is just a number, the result is an NxN matrix A, whose each element A_{ij} is the dot product of the i-th query vector with the j-th key vector.

This process is neatly summarized by the equation:

$$Attention(Q, K, V) = softmax\left(\frac{Q.K^T}{\sqrt{d_k}}\right).V$$

The dot product between Keys and Queries vectors can yield quite large, or small, numbers, we can interpret this as the relevance level representation of each word. To have numerical stability and still keep the relevance level as we intend, we have to "normalize" these values. Thus, the SoftMax function, but before we apply the SoftMax function, we need to make sure that future words do not affect the attention matrices for the past words, since this would render the task of predicting next word useless. To do this we apply masking (setting to $-\infty$) all inputs that gives this problem (upper diagonal elements).

Points to note:

- In the vanila self-attention mechanism, each word (token) interacts with every other, there is no temporal ordering in the inputs.
- The computational complexity of calculating the attention matrix is N^2. Hence this "arms" race for bigger hardware and the increasing in the number of tokens that can be processed by the model.
- the ‘self’ in self-attention refers to the fact that we also take the dot product of a given word’s query with its own key vector.
- We also have cross-attention, which is commonly used in translations, text-to-speech and vision transformers models, usually models that make use of two (or more) different types of inputs.

Self-Attention Process

3. Attend: We take the dot product of the normalized attention scores and the value vectors.

$$softmax\left(\frac{Q.K^T}{\sqrt{d_k}}\right).V$$

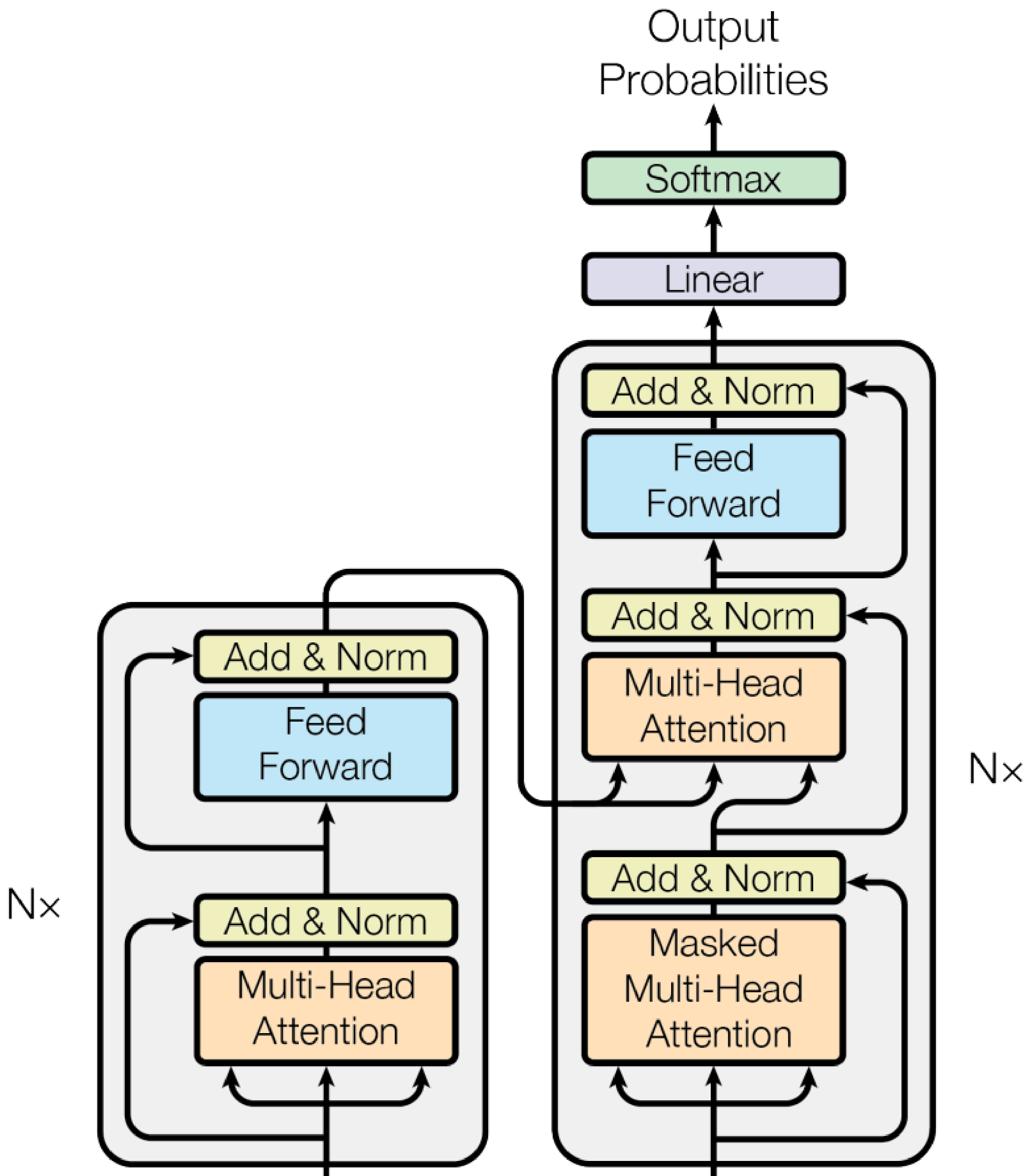
Since the value vector V are encoded for each word using a dense layer, the output of the attention mechanism has N vectors, the same as the number of input words. Finally, to obtain the output of the self-attention layer, we transform the *Attention(Q,K,V)* vector into the same size string as the input with a dense layer. These value vectors can be seen as a correction factor to the relevance of the predicted words, in essence is like a course predictions for the vectors of the next word. It can also see as a ranking system for the most likely word in the sequence.

During the training phase, the model would try to predict the next word of the sequence in a manner of first start with small sequences of tokens and gradually increasing it

- I -> ???
- I put -> ???
- I put on -> ???
- I put on a -> ???
- I put on a light -> ???

Also at training time, the model would process larger batch sizes (512) vs. the batch size of one that evaluation uses.

There we have - The transformer model, Ladies and Gentlemen


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx

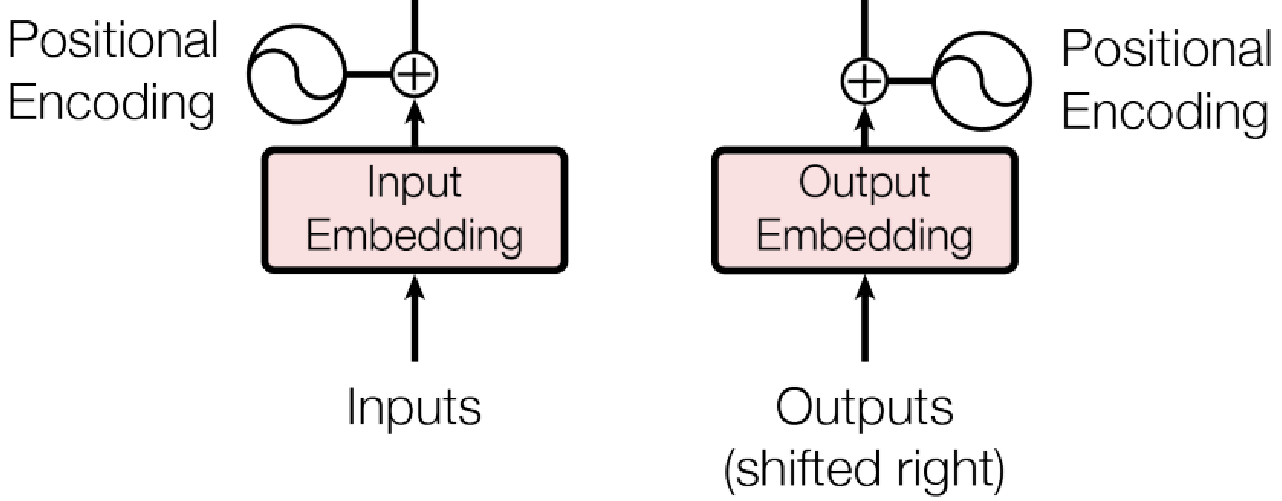


Figure 1: The Transformer - model architecture.

Ofcourse, there was quite a lot over-simplefications and I skipped to a lot of details, suxh as:

- multi-head attention: Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

- Transformers use a lot of layer normalization, did not mentioned but it is used quite a lot in the architecture.
- Did not mention temperature, but it is another component of the self-attention mechanism that helps the performance of the model

From Transformers to Generative Pre-trained Transformers (GPT) and beyond

The “attention is all you need” paper that gave birth to transformers, was still focused on the problem of translation trained in a supervised way.

Meanwhile, Open AI saw the result and immediately tried this more powerful transformer architecture on the next word prediction problem at a larger scale not before possible.

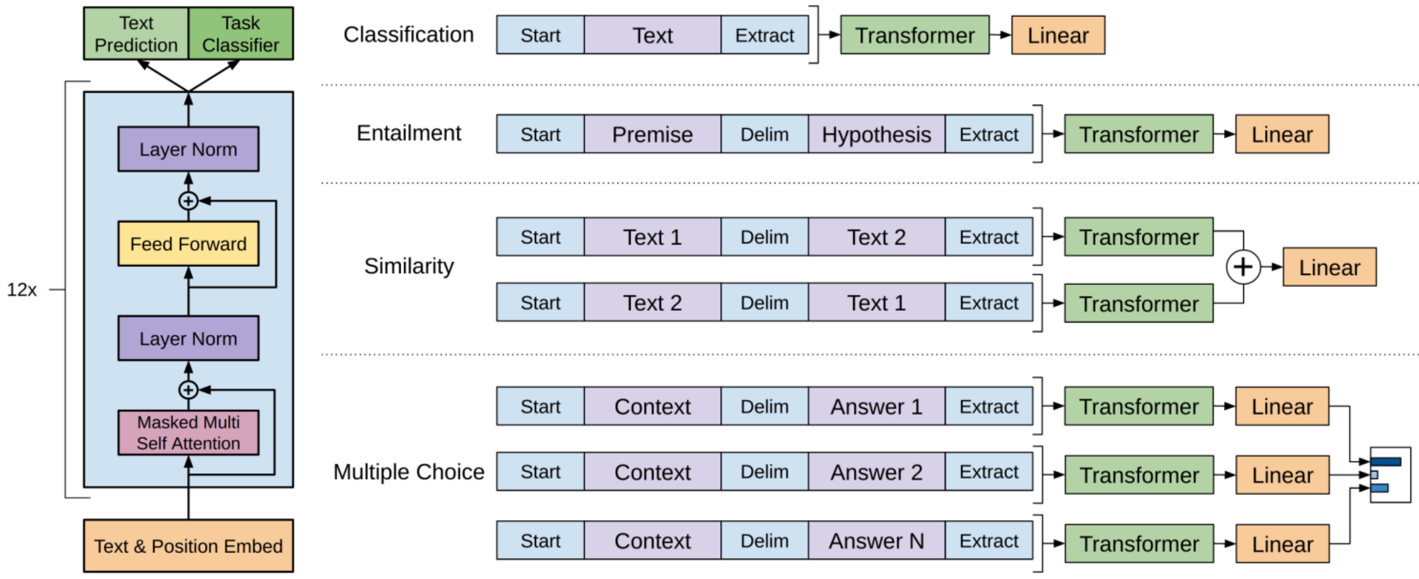


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

They split the training in two stages:

1. learning a high-capacity language model on a large corpus of text.
2. a fine-tuning stage, where we adapt the model to a discriminative task with labelled data.

And the results found was nothing shorter than surprisingly, as most important:

- Whem prompted with text segments it would continue with more coerent text -> see by yourself
- It showed some capability in answering general questions.
- It also exhibits zero-shot learning behaviour, the model was capable of answering questions not present in the training data.

Zero-shot Behaviors: "We'd like to better understand why language model pre-training of transformers is effective. A hypothesis is that the underlying generative model learns to perform many of the tasks we evaluate on in order to improve its language modeling..."

From Transformers to Generative Pre-trained Transformers (GPT) and beyond

The success of GPT-1 gave a boost on the incentive to build larger models and training in even bigger datasets, so OpenAI quickly follow with GPT-2 (~ 1.5 billion of parameters) and GPT-3 (175 billion of parameters).

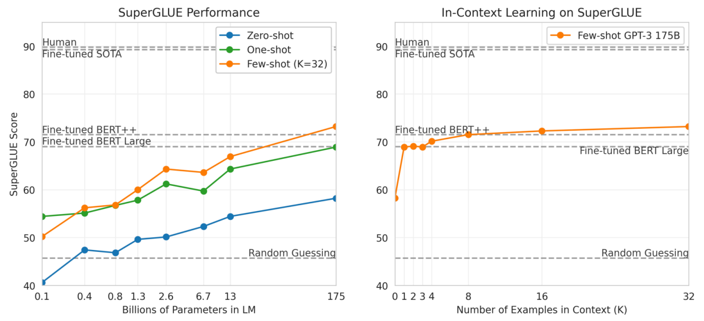
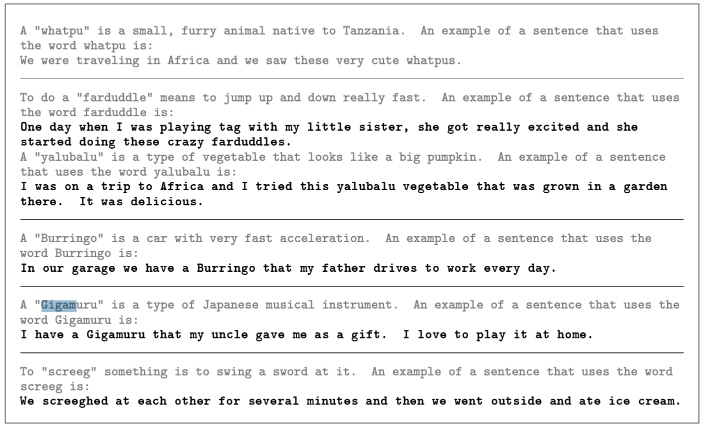


Figure 3.8: Performance on SuperGLUE increases with model size and number of examples in context. A value of $K = 32$ means that our model was shown 32 examples per task, for 256 examples total divided across the 8 tasks in SuperGLUE. We report GPT-3 values on the dev set, so our numbers are not directly comparable to the dotted reference lines (our test set results are in Table 3.8). The BERT-Large reference model was fine-tuned on the SuperGLUE training set (125K examples), whereas BERT++ was first fine-tuned on MultiNLI (392K examples) and SWAG (113K examples) before further fine-tuning on the SuperGLUE training set (for a total of 630K fine-tuning examples). We find the difference in performance between the BERT-Large and BERT++ to be roughly equivalent to the difference between GPT-3 with one example per context versus eight examples per context.



```
A "whatpu" is a small, furry animal native to Tanzania.  An example of a sentence that uses
the word whatpu is:
We were traveling in Africa and we saw these very cute whatpus.
```

```
To do a "farduddle" means to jump up and down really fast.  An example of a sentence that uses
the word farduddle is:
```
One day when I was playing tag with my little sister, she got really excited and she started doing these crazy farduddles.
```
A "yalubalu" is a type of vegetable that looks like a big pumpkin.  An example of a sentence
that uses the word yalubalu is:
```
I was on a trip to Africa and I tried this yalubalu vegetable that was grown in a garden there. It was delicious.

```
A "Burringo" is a car with very fast acceleration.  An example of a sentence that uses the
word Burringo is:
```
In our garage we have a Burringo that my father drives to work every day.

```
A "Gigamuru" is a type of Japanese musical instrument.  An example of a sentence that uses the
word Gigamuru is:
```
I have a Gigamuru that my uncle gave me as a gift. I love to play it at home.

```
To "screeg" something is to swing a sword at it.  An example of a sentence that uses the word
screeg is:
```
We screeghed at each other for several minutes and then we went outside and ate ice cream.

Figure 3.16: Representative GPT-3 completions for the few-shot task of using a new word in a sentence. Boldface is GPT-3's completions, plain text is human prompts. In the first example both the prompt and the completion are provided by a human; this then serves as conditioning for subsequent examples where GPT-3 receives successive additional prompts and provides the completions. Nothing task-specific is provided to GPT-3 other than the conditioning shown here.

GPT-3 showed an increase performance in all metrics tested (zero-shot, text comprehension, ...), but one capability really jumped out. Once training was complete you could still teach the network new things. This is also know as **in context learning** also called “zero-shot transfer”.

Lets build a GPT-2 model

```
In [1]: import torch
        import numpy as np
        import matplotlib.pyplot as plt

        from torch import nn
        from torch.optim import Adam

        from tqdm import tqdm

        from sklearn.metrics import confusion_matrix
        from sklearn.metrics import ConfusionMatrixDisplay
```

```
In [2]: from transformers import GPT2Model, GPT2Tokenizer
```

```
2024-04-18 10:40:16.460544: I tensorflow/core/util/port.cc:110]
oneDNN custom operations are on. You may see slightly different
numerical results due to floating-point round-off errors from d
ifferent computation orders. To turn them off, set the environm
ent variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-04-18 10:40:16.534527: I tensorflow/core/platform/cpu_feat
ure_guard.cc:182] This TensorFlow binary is optimized to use av
ailable CPU instructions in performance-critical operations.
```

```
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 AV
X512F AVX512_VNNI FMA, in other operations, rebuild TensorFlow
with the appropriate compiler flags.
```

In [3]:
```
import pandas as pd
```

Dataset and task

For this example we are going to use the BBC news classification dataset, comprised of 2225 articles, each labeled under one of 5 categories: business, entertainment, politics, sport or tech.:

- BBC News Train.csv - the training set of 1490 records
- BBC News Test.csv - the test set of 736 records
- BBC News Sample Solution.csv - a sample submission file in the correct format

The task is quite simple: just classify if a given sentence belongs to one of the 5 categories:

Loading the model into a dataframe:

In [4]:
```
df = pd.read_csv("datasets/bbc_data/BBC News Train.csv")
```

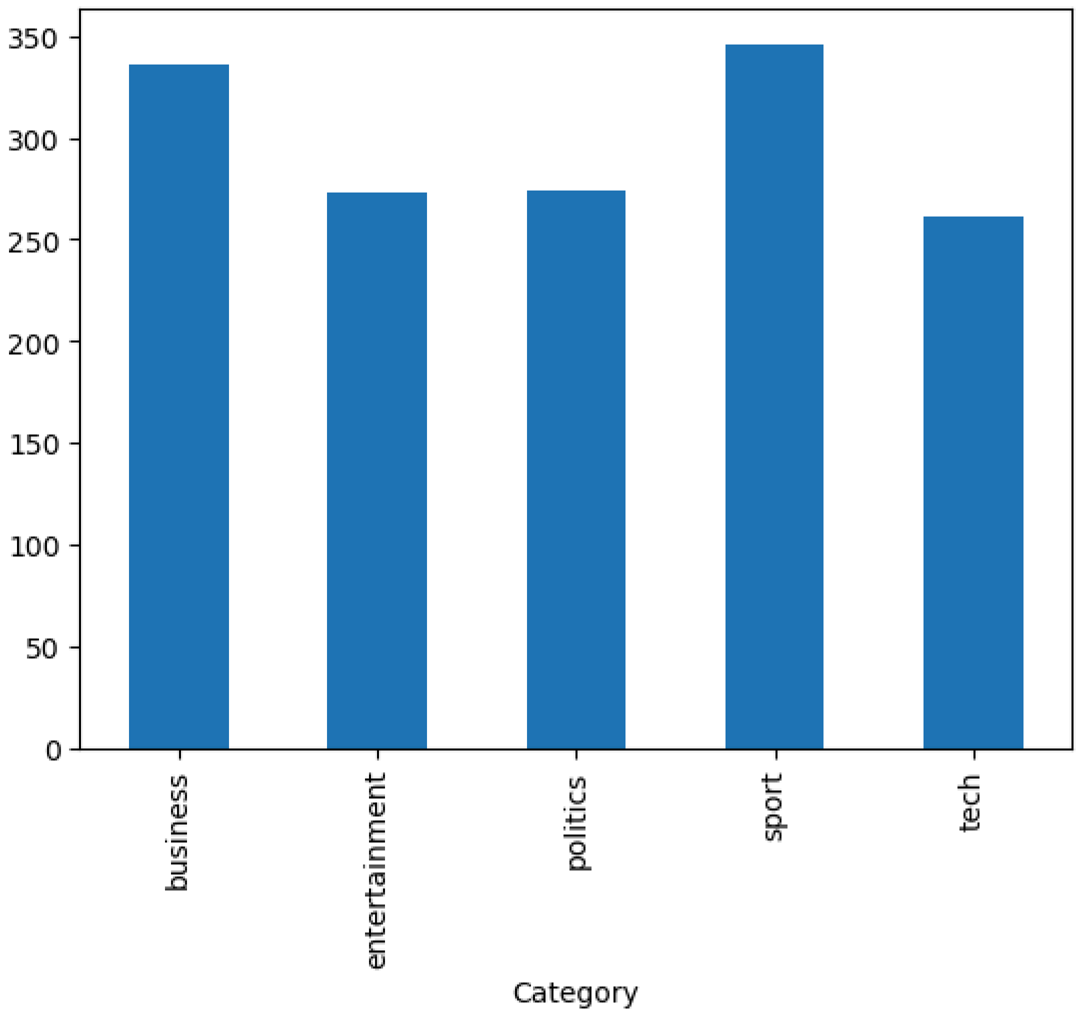
Checking the dataframe

In [5]:
```
df.head(5)
```

Out[5]:

	ArticleId	Text	Category
0	1833	worldcom ex-boss launches defence lawyers defe...	business
1	154	german business confidence slides german busin...	business
2	1101	bbc poll indicates economic gloom citizens in ...	business
3	1976	lifestyle governs mobile choice faster bett...	tech
4	917	enron bosses in $168m payout eighteen former e...	business

```
In [6]: df.groupby("Category").size().plot.bar()

Out[6]: <Axes: xlabel='Category'>
```



Preparing the building blocks: starting from the tokenizer

```
In [7]: tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
```

```
In [9]: tokenizer.padding_side = "left"
        tokenizer.pad_token = tokenizer.eos_token
```

Obs: Some models are trained with left or right padding, in this case our model is trained with left padding, but other models and even trained in other languages a right padded model might be prefeered.

Encoding a sentence

```
In [10]: example_text = "i don't like sand it's coarse and rough and gets everyw
```

```
In [11]: gpt2_input = tokenizer(example_text, padding="max_length", max_length=2
```

```
In [12]: gpt2_input
```

```
Out[12]: {'input_ids': tensor([[50256, 50256, 50256, 50256, 50256, 5025
         6, 50256,    72,   836,   470,
                   588,  6450,   340,   338, 36076,   290,  5210,   29
         0,  3011,  8347]]), 'attention_mask': tensor([[0, 0, 0, 0, 0,
         0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])}
```

Decoding

```
In [13]: example_text = tokenizer.decode(gpt2_input.input_ids[0])
         print(example_text)
```

```
<|endoftext|><|endoftext|><|endoftext|><|endoftext|><|endoftext
|><|endoftext|><|endoftext|>i don't like sand it's coarse and r
ough and gets everywhere
```

Wrapping our tokenizer

In [14]:
```
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
tokenizer.padding_side = "left"
tokenizer.pad_token = tokenizer.eos_token
labels = {
    "business": 0,
    "entertainment": 1,
    "sport": 2,
    "tech": 3,
    "politics": 4
         }
```

Building a dataset

Usually when using pytorch we create a dataset object which expose the data to our model to be trained, is a simple class that nowadays are automagically built by some wrappers, like we did with ultralytics, or we can build our own custom dataset, we just need to implement a **class** that contains the following 3 methods:

- `__init__` : to kick start our dataset object when its called, We initialize the dataframe containing the **features** and **targets** of our dataset
- `__len__`: to returns the number of samples in our dataset.

and

- `__getitem__`: to loads and returns a sample from the dataset at the given index idx

In [15]:
```
class Dataset(torch.utils.data.Dataset):
    def __init__(self, df):
        self.labels = [labels[label] for label in df['Category']]
        self.texts = [tokenizer(text,
                                padding='max_length',
                                max_length=128,
                                truncation=True,
                                return_tensors="pt") for text in df['Te

    def classes(self):
        return self.labels

    def __len__(self):
        return len(self.labels)

    def get_batch_labels(self, idx):
        # Get a batch of labels
        return np.array(self.labels[idx])

    def get_batch_texts(self, idx):
        # Get a batch of inputs
        return self.texts[idx]

    def __getitem__(self, idx):
        batch_texts = self.get_batch_texts(idx)
        batch_y = self.get_batch_labels(idx)
        return batch_texts, batch_y
```

```
In [16]: np.random.seed(42)
         df_train, df_val, df_test = np.split(df.sample(frac=1, random_state=35)
                                                      [int(0.8*len(df)), int(0.9*len(df)

         print(len(df_train), len(df_val), len(df_test))
```

```
1192 149 149
```

```
/opt/conda/lib/python3.10/site-packages/numpy/core/fromnumeric.
py:59: FutureWarning: 'DataFrame.swapaxes' is deprecated and wi
ll be removed in a future version. Please use 'DataFrame.transp
ose' instead.
  return bound(*args, **kwds)
```

Building our model

For our task we need to modify a little bit the original gpt-2 model, luck for us that is not a diffucult task. We just need first load the original model by doing:

```
GPT2Model.from_pretrained(gpt_model_name)
```

and latter we create a classification head using a dense layer where we pass the final tensors from the gpt-2 model and outputs the logits for each one of the categories

```
In [19]: class SimpleGPT2SequenceClassifier(nn.Module):
             def __init__(self, hidden_size: int, num_classes:int ,max_seq_len:i
                 super(SimpleGPT2SequenceClassifier,self).__init__()
                 self.gpt2model = GPT2Model.from_pretrained(gpt_model_name)
                 self.fc1 = nn.Linear(hidden_size*max_seq_len, num_classes)

             def forward(self, input_id, mask):
                 """
                 Args:
                         input_id: encoded inputs ids of sent.
                 """
                 gpt_out, _ = self.gpt2model(input_ids=input_id, attention_mask=
                 batch_size = gpt_out.shape[0]
                 linear_output = self.fc1(gpt_out.view(batch_size,-1))
                 return linear_output
```

Obs**

One very common mistake while we re-using models is the dimension shapes of the inputs and outputs, a tip for this problem is to use as best as you can the `view`, `reshape` and other reshape functions that manipulate the tensors. Just keep in mind that some of these functions create new copies of the tensors, which can put a price on memory space and overheads.

The training and test loops

```
In [20]: def train(model, train_data, val_data, learning_rate, epochs):
             train, val = Dataset(train_data), Dataset(val_data)

             train_dataloader = torch.utils.data.DataLoader(train, batch_size=2,
             val_dataloader = torch.utils.data.DataLoader(val, batch_size=2)

             use_cuda = torch.cuda.is_available()
             device = torch.device("cuda" if use_cuda else "cpu")

             criterion = nn.CrossEntropyLoss()
             optimizer = Adam(model.parameters(), lr=learning_rate)

             if use_cuda:
                 model = model.cuda()
                 criterion = criterion.cuda()

             for epoch_num in range(epochs):
                 total_acc_train = 0
                 total_loss_train = 0

                 for train_input, train_label in tqdm(train_dataloader):
                     train_label = train_label.to(device)
                     mask = train_input['attention_mask'].to(device)
                     input_id = train_input["input_ids"].squeeze(1).to(device)

                     model.zero_grad()
```

```
            output = model(input_id, mask)

            batch_loss = criterion(output, train_label)
            total_loss_train += batch_loss.item()

            acc = (output.argmax(dim=1)==train_label).sum().item()
            total_acc_train += acc

            batch_loss.backward()
            optimizer.step()

        total_acc_val = 0
        total_loss_val = 0

        with torch.no_grad():

            for val_input, val_label in val_dataloader:
                val_label = val_label.to(device)
                mask = val_input['attention_mask'].to(device)
                input_id = val_input['input_ids'].squeeze(1).to(device)

                output = model(input_id, mask)

                batch_loss = criterion(output, val_label)
                total_loss_val += batch_loss.item()

                acc = (output.argmax(dim=1)==val_label).sum().item()
                total_acc_val += acc

            print(
            f"Epochs: {epoch_num + 1} | Train Loss: {total_loss_train/l
            | Train Accuracy: {total_acc_train / len(train_data): .3f}
            | Val Loss: {total_loss_val / len(val_data): .3f} \
            | Val Accuracy: {total_acc_val / len(val_data): .3f}")
```

In [21]:
```python

EPOCHS = 3
model = SimpleGPT2SequenceClassifier(hidden_size=768, num_classes=5, ma
LR = 1e-5
```

In [22]:
```python
train(model, df_train, df_val, LR, EPOCHS)
```

```
100%|██████████| 596/596 [00:50<00:00, 11.87it/s]
```

```
Epochs: 1 | Train Loss:  0.302            | Train Accuracy:
0.794            | Val Loss:  0.207            | Val Accurac
y:  0.899
```

```
100%|██████████| 596/596 [00:50<00:00, 11.85it/s]
```

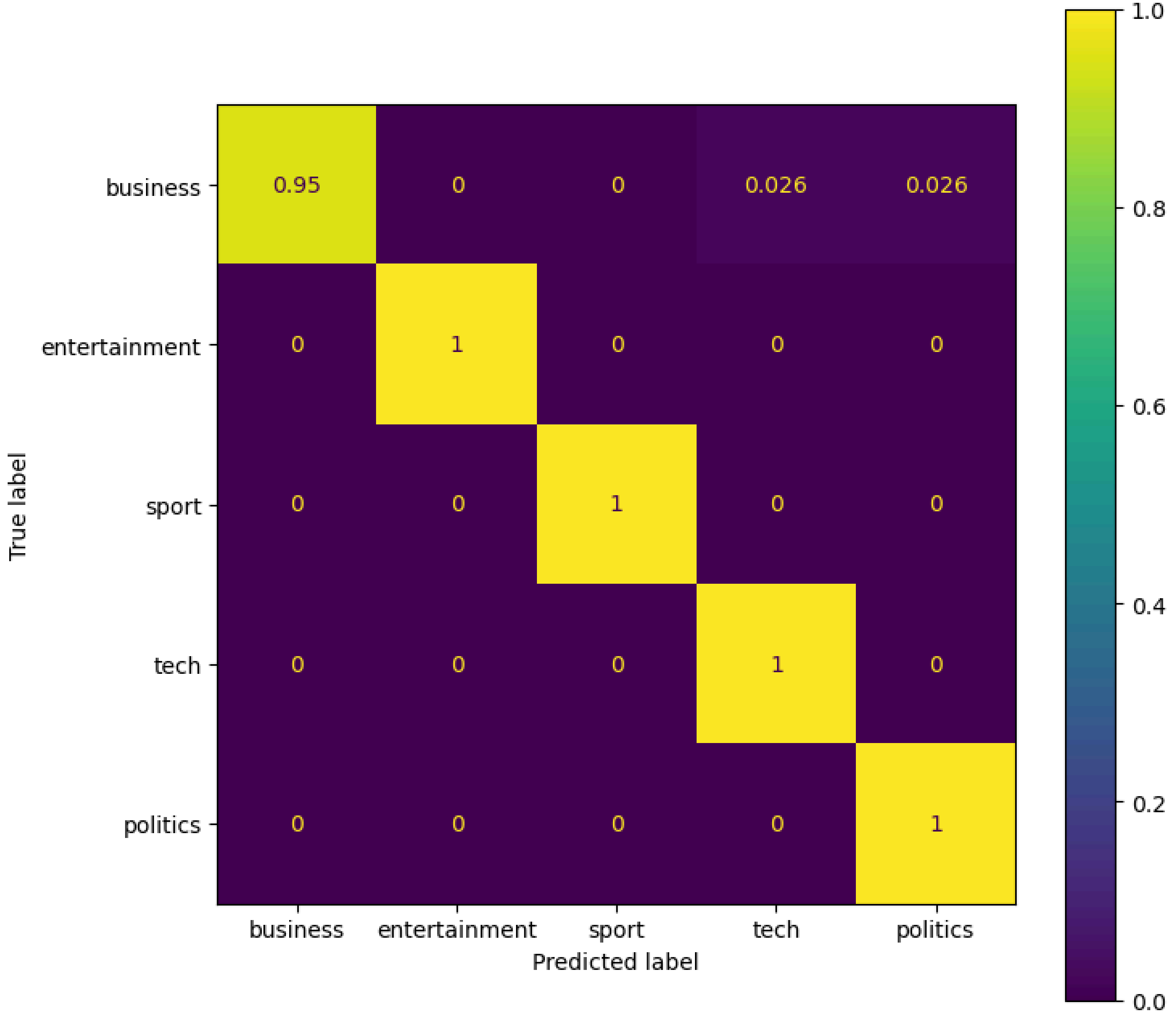
```
Epochs: 2 | Train Loss:  0.038            | Train Accuracy:
0.975            | Val Loss:  0.099            | Val Accurac
y:  0.953
```

```
100%|██████████| 596/596 [00:50<00:00, 11.81it/s]
```

```
Epochs: 3 | Train Loss:  0.008            | Train Accuracy:
0.996            | Val Loss:  0.125            | Val Accurac
y:  0.960
```

In [23]:
```
def evaluate(model, test_data):

    test = Dataset(test_data)

    test_dataloader = torch.utils.data.DataLoader(test, batch_size=2)

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    if use_cuda:

        model = model.cuda()


    # Tracking variables
    predictions_labels = []
    true_labels = []

    total_acc_test = 0
    with torch.no_grad():

        for test_input, test_label in test_dataloader:

            test_label = test_label.to(device)
            mask = test_input['attention_mask'].to(device)
            input_id = test_input['input_ids'].squeeze(1).to(device)

            output = model(input_id, mask)

            acc = (output.argmax(dim=1) == test_label).sum().item()
            total_acc_test += acc

            # add original labels
            true_labels += test_label.cpu().numpy().flatten().tolist()
            # get predicitons to list
            predictions_labels += output.argmax(dim=1).cpu().numpy().fl

    print(f'Test Accuracy: {total_acc_test / len(test_data): .3f}')
    return true_labels, predictions_labels
```

```
In [24]: true_labels, pred_labels = evaluate(model, df_test)

         Test Accuracy:  0.987

In [25]: # Plot confusion matrix.
         fig, ax = plt.subplots(figsize=(8, 8))
         cm = confusion_matrix(y_true=true_labels, y_pred=pred_labels, labels=ra
         disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=list(
         disp.plot(ax=ax)

Out[25]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay
         at 0x7f32faab3e20>
```


business
entertainment
sport
tech
politics
True label
Predicted label
0.95
0
0
0.026
0.026
0
1
0
0
0
0
0
1
0
0
0
0
0
1
0
0
0
0
0
1
1.0
0.8
0.6
0.4
0.2
0.0