Member-only story

2D Tokenization for Large Language Models

This article is about how we process text before passing it to large language models, the problems with this approach, and an alternative solution.

David Gilbertson · Follow

Published in Better Programming · 18 min read · May 2

The Problem With (1D) Tokenization

When passing text to a Large Language Model (LLM), text is broken down into a sequence of words and sub-words. This sequence of tokens is then replaced with a sequence of integers and passed to the model.

LLMs contain an embedding matrix to store a representation for each of these tokens. In the case of the RoBERTa model, there are 768 numbers to represent each of the ~50,000 tokens in its vocabulary.

This approach raises several questions, like *"how should spaces be represented in the sequence of tokens?"* and *"should different capitalization be considered a different word?"*.

If we look in the embedding matrix of the RoBERTa model, there isn't just one representation of the word `dog`, there are separate representations for variations in capitalization, pluralization, and whether or not it's preceded by a space. In total, RoBERTa has seven separate slots in its vocab for various forms of `dog`.

```
>>> roberta_tok.convert_ids_to_tokens([39488, 8563, 16319, 2335, 3678, 18619, 20
['Dog', 'ĠDog', 'dog', 'Ġdog', 'Ġdogs', 'ĠDogs', 'dogs']
```

There is no mechanism for the model to consider the tokens as related in any way; every time it learns something new about our furry little friends, it will update **only one** of these seven embeddings.

And if the problem is bad for nouns, it's even worse for verbs. There are 15 variants of `run` in RoBERTa's 50k token vocabulary. On top of this, a word like `riding` (which isn't in the vocabulary) will be split, but of course won't be split into `ride` and `ing`, instead, it is split into `r` and `iding`.

It can't be easy for the model to learn when different tokens represent the same real-world thing; especially considering they often don't occur in the same document (or gradient update step). To put yourself in the model's shoes, imagine trying to learn a language where the seven variants of `dog` listed above bore no resemblance to one another. Possible, but slow going.

2D Tokenization

The idea of 2D tokenization is to pass similar information to the model, but represented as a root word token and a set of binary 'flags', so that the model

can maintain an embedding for the root word and also know what variant of the word it's dealing with.

Take a look at this sentence: **I enjoyed riding in Australia's parks.**

This can be represented in 2D form like so:

Root	Token ID	Capital	Plural	-'s	-ing	-ed
i	77	TRUE	FALSE	FALSE	FALSE	FALSE
enjoy	4173	FALSE	FALSE	FALSE	FALSE	TRUE
ride	2173	FALSE	FALSE	FALSE	TRUE	FALSE
in	262	FALSE	FALSE	FALSE	FALSE	FALSE
australia	6367	TRUE	FALSE	TRUE	FALSE	FALSE
park	1342	FALSE	TRUE	FALSE	FALSE	FALSE

The model can learn what a `park` is and learn what it means to `ride` and also learn what `ing` does and what `'s` means. This should allow a richer representation, a more condensed vocabulary, and faster learning.

At least, that's what I had hoped…

Project status

Alas, I couldn't get it to work (more specifically, I couldn't get it to outperform plain RoBERTa). But I want to write about it anyway, for a few reasons:

1. I'm new to all this — a one-year-old in the Python/Data Science/Machine Learning world, so it's quite possible that my ideas are sound but my

ability to execute is lacking. In other words, maybe I'm doing something dumb and someone more capable than me will point it out.

2. Maybe this approach sparks an idea and sets someone else down the path of creating another solution.
3. Negative results are still results.
4. Some light reading for potential future employers.

Below, I'll describe all the components of the system, then get to the (non) results.

I'm comparing this approach against RoBERTa, and will refer to my 2D variants as R2D (**R**oBERTa, with **2D** inputs).

The deconstruction game

To start, I needed to be able to break certain words apart into their root, and a representation of the variation from the root. (This step is optional, as I'll explain in a later section).

My first attempt didn't use a 2D representation, it simply pre-processed text to break words apart, turning `riding` into `ride ~ing`, `unbeatable` into `un~ beat ~able`, etc.

I started with the OpenWebText corpus of 8 million web pages and generated a word frequency list. Luckily word usage distributions are far from linear, so I only needed to go over a few thousand words to learn if the idea showed any promise.

For this task, I wrote a little CLI-based 'game' that would loop over these words, in order from most common to least. For each word, it would have a rough guess at what some 'deconstructed' forms might be, if any, and present me with the word and those options. I could then choose to either keep the word as is, accept one of the suggested deconstructions, or type my own.

The game is to see how many you can do in one hour, and then try and beat that in the next hour.

In the picture below, text to the left of the green `>?` is what the game presented to me, and text to the right is my response (blank means the word should not be deconstructed).

```
[imagine]>?
[immigration] (a: immigra ~tion)>? immigrate ~tion
[enforcement] (a: enforce ~ment)>? a
[communities] (a: community ~s) (b: communitie ~s)>? a
[housing] (a: hous ~ing) (b: house ~ing) (c: hou ~ing)>? b
[planned] (a: plann ~ed) (b: planne ~ed) (c: plan ~ed)>? c
[quick]>?
```

Some words can technically be split but perhaps shouldn't be. So when playing the deconstruction game, the question I asked myself for each possibly-deconstructable word was: *Does the root of this word share significant meaning with the variant?*

For example, does the concept of `editor` build upon the concept of `edit`?

I played the game for the top 4,700 words, 2,000 of which could be deconstructed. This covers about 80% of the corpus. Here's the list.

As it turns out, I didn't end up using a lot of the affixes (as flags) because they were so sparse that training didn't proceed. From a list of 50+ affixes (`-tion`, `dis-`, `-ment`, etc.) I only ended up using the four most common in the main experiments below.

Even though I later moved on to a 2D representation, I've kept the deconstructions defined with this tilde format, which is easy enough to parse to get the root word and various affixes as flags. (This format would not be suitable for many other languages.)

All in all I only spent a few hours playing the deconstruction game, a tiny fraction of the seven weeks spent tinkering with the rest of the system.

I think any attempt to automate this is ultimately doomed. Our language is far too sloppy. For example, `documentary` could *technically* be split, but it would be a stretch to claim that this is a variant of the word `document` in any way that would help an LLM learn. Same for `disarming`, `swimmingly`, `treatment`, `difference`, `behaviour`, `handed`, `founded`, `refund`, and many, many more. This is also an answer to the question "why didn't you just use such-and-such a tool?". Certainly, others have done this work, but from what I could find they are all based on linguistic technicalities, not on what will help a language model learn (shared meaning between root and variant).

Now is a good time to note that I'm well aware of the The Bitter Lesson and sensibly wary of any attempt to help models learn by manually encoding the rules of language. And maybe tokenizer-free models like CANINE and ByT5 are the future, but the goal of R2D is to improve upon the 1D tokenization

approach that throws away so much information. Specifically, to maintain the relationships between words that are apparent to us humans when reading them on the page (if you've seen `dog`, you know what `dogs` means even if you've never seen it before), but not apparent after they've been reduced to numbers (if you've seen `16319`, it doesn't follow that you know `20226` just means many of `16319`).

So if you too are wary of the bitter lesson, think of this more as an attempt to *throw away less information* than it is an attempt to manually encode additional information.

The R2D transform

For my dataset, tokenizer, and model, I'm leveraging Hugging Face. In the Hugging Face world, a transform processes data from the dataset before passing it to the model. The R2D transform takes text as input and outputs a numerical 2D representation ready to be passed to the model. (As with most components, it operates on batches, but for clarity, I'll leave those out of the descriptions).

It does this in two steps:

1. Convert a text sequence to 2D form, with root words and flags
2. Tokenize the root words, add special tokens, pad, and truncate

Step 1: converting sequences to 2D

The output of this step looks like so:

Root	NoSpace	FirstCap	AllCaps	Suffix_s	Suffix_ed	Suffix_ing	Contraction_s
i	True	True	False	False	False	False	False
enjoy	False	False	False	False	True	False	False
ride	False	False	False	False	False	True	False
in	False	False	False	False	False	False	False
australia	False	True	False	False	False	False	True
park	False	False	False	True	False	False	False

I'm using 7 flags here, but this could be up to ~60

The `NoSpace` flag indicates that there was not a space before the word. This is the inverse of the `Ġ` prefix that RoBERTa adds, which indicates that there *was* a space before the token. Personally, I find this an odd thing to track because it treats the following things as the same: a) words at the start of a document, b) words after a `(` or `'` or `"`, c) word parts that resulted from splitting a token into sub-word tokens. But since I'm comparing R2D with RoBERTa, I have kept this logic as is.

The input text is split into words using a RoBERTa tokenizer that I've modified to *not* split apart `'s`. More correctly, this uses the *pre-tokenizer* — the part of the tokenizer that splits sequences, not the part that has been trained on a corpus to build a vocabulary.

```
>>> roberta_split("I enjoyed riding in Australia's parks")
['I', 'Ġenjoyed', 'Ġriding', 'Ġin', "ĠAustralia's", 'Ġparks']
```

For each of these words, I get the root and assign flags. The values for the flags come from two places:

1. Inferring flags directly from the strings (`NoSpace`, `FirstCap`, `AllCaps`, `Contraction_s`)

2. Looking up the word in the list of deconstructions (`Suffix_s` , `Suffix_ed` , `Suffix_ing` , `Contraction_s`). If a word is in this list, I get the root from the list.

As you may have noticed, this means you could implement this whole system without the manually created list of deconstructions, if you only wanted the flags `NoSpace` , `FirstCap` , `AllCaps` and `Contraction_s` . This is what I meant earlier when I said that the Deconstruction Game was optional.

This step of the transform will also combine multiple input sequences, such as the `sentence1` / `sentence2` pairs found in the GLUE dataset.

Note: it's perhaps a bit confusing, but step 1 of the transform is required in order to train the R2D tokenizer (because it's only shown the root words), and the R2D tokenizer is required to complete step 2 of the transform.

Step 2: tokenize, pad, truncate

Step 2 takes as input the output from step 1, believe it or not. It outputs something like this:

TokenID	NoSpace	FirstCap	AllCaps	Suffix_s	Suffix_ed	Suffix_ing	Contraction_s
0	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000
77	0.5000	0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000
4173	-0.5000	-0.5000	-0.5000	-0.5000	0.5000	-0.5000	-0.5000
2173	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	0.5000	-0.5000
262	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000
6367	-0.5000	0.5000	-0.5000	-0.5000	-0.5000	-0.5000	0.5000
1342	-0.5000	-0.5000	-0.5000	0.5000	-0.5000	-0.5000	-0.5000
2	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000
...	...	...	...	...	...	...	...
1	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000
1	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000
1	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000	-0.5000

The words from Step 1 are extracted and passed to the R2D tokenizer. If a word is split into multiple tokens, extra rows are inserted into the table.

Also, the special tokens for the start and end of a sequence are added as rows, with all flags set to `False`.

If the total number of rows falls short of the maximum sequence length the model can accept (e.g. 512 tokens) then extra rows are added at the end with the padding token ID (`1`) and `False` for all the flags.

If there are too many rows, they are truncated.

Lastly, the Boolean values are converted to `-0.5` / `0.5` ready for the model. (Later tests revealed that `0` / `1` is just as good — and a lot easier to read when debugging. I guess in that case some normalization layer takes care of the non-zero mean.)

The R2D tokenizer

This is a Hugging Face brand tokenizer. It's the same as a RoBERTa tokenizer, but is trained on the root words that come out of step 1 of the 2D transform process described above. That is, its vocab will only have the word `dog`, not `Dog`, `dog's`, `dogs`, etc.

After the top few thousand most common words, about 50% of words are variations of a root word (this surprised me!). So the R2D tokenizer packs quite a few more 'real words' into its 50k vocabulary than the plain RoBERTa tokenizer.

Of the top 50,000 English words (from a sampling of the OpenWebText corpus, all lower case):

- 22,037 are in the RoBERTa tokenizer vocab (after allowing for case and removing `Ġ`)
- 34,727 are in the R2D tokenizer vocab (or their root is)

That's 58% more words that can be represented by a single token. This means that important words — like `permalink`, `bison` and `mandlebrot` — get their own slot in the R2D model's embedding matrix. This is not a huge deal, because a bigger vocab doesn't add much to the model size, but worth mentioning.

The R2D model

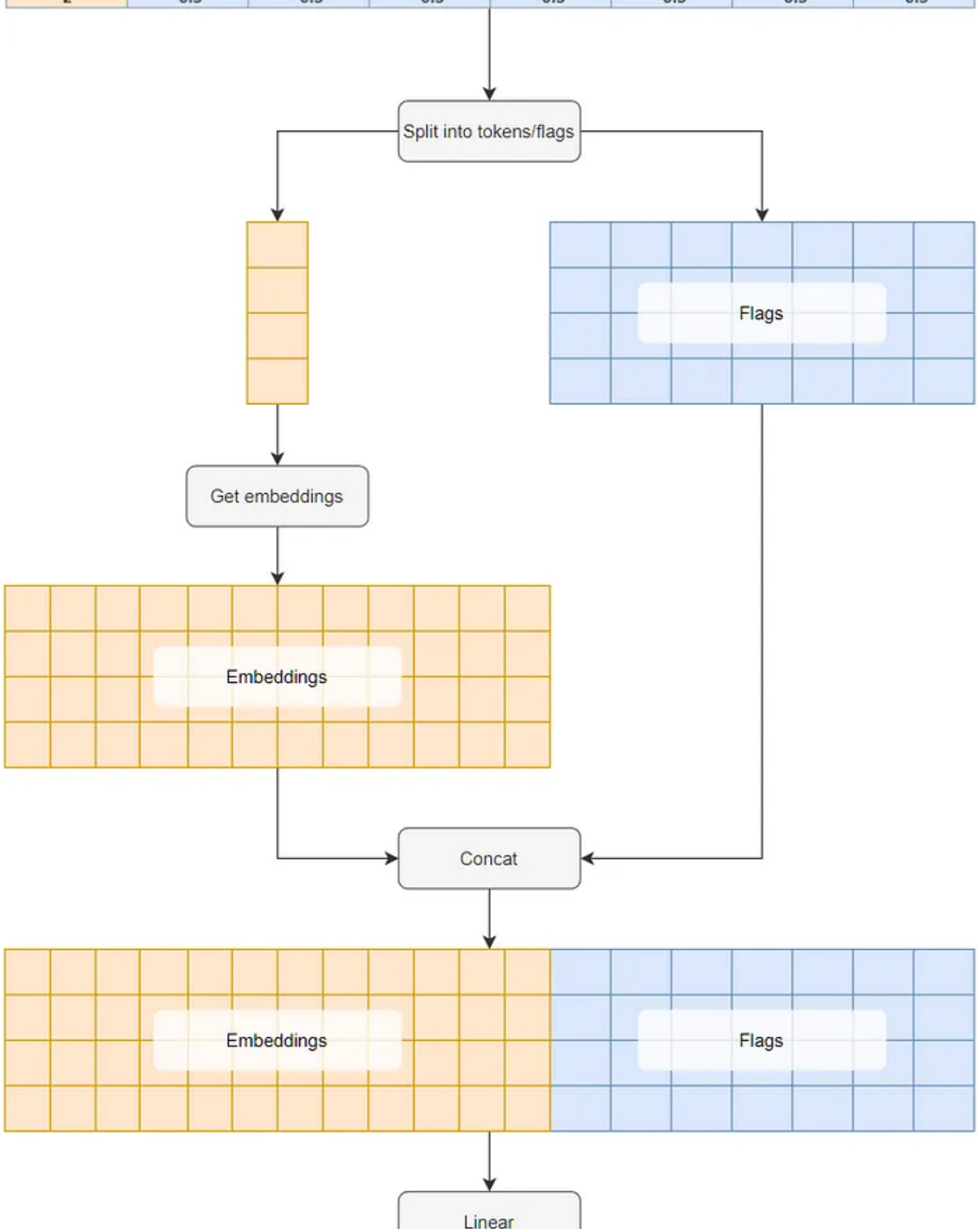
The R2D model is an extension of RoBERTa. I chose RoBERTa for its relative popularity, familiarity, and simplicity. There are no changes to the guts of the model, only some preparation steps at one end and an extra head at the other.

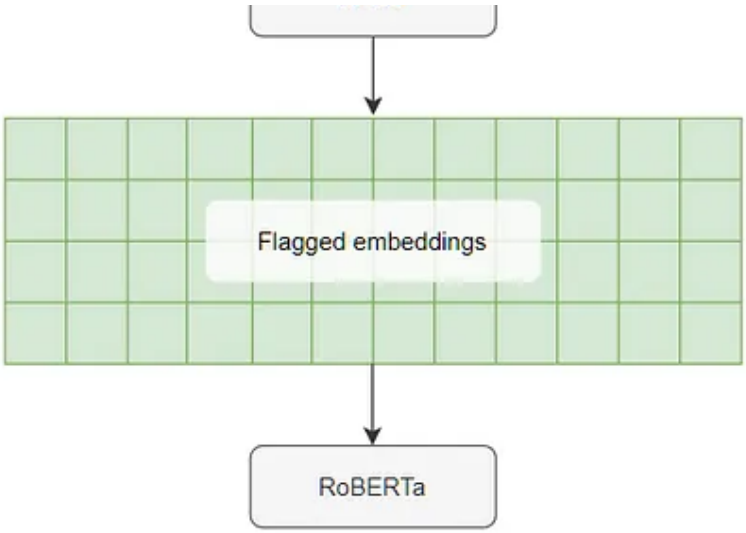
Preparation steps

Before data is sent to the attention layers of the model:

- The 2D input is split into separate tensors of **token IDs** and **flags**
- The token IDs are converted to their embedding vectors
- The embeddings are concatenated with the flags and condensed down to the correct size (512 x 768)

TokenID	NoSpace	FirstCap	AllCaps	Suffix_s	Suffix_ed	Suffix_ing	Contraction_s
0	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5
77	0.5	0.5	-0.5	-0.5	-0.5	-0.5	-0.5
4173	-0.5	-0.5	-0.5	-0.5	0.5	-0.5	-0.5
2173	-0.5	-0.5	-0.5	-0.5	-0.5	0.5	-0.5
262	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5
6367	-0.5	0.5	-0.5	-0.5	-0.5	-0.5	0.5
1342	-0.5	-0.5	-0.5	0.5	-0.5	-0.5	-0.5
2	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5	-0.5





Showing my newbie status by making diagrams with a top-to-bottom flow.

As you can see, even though the model's embedding matrix will only have a single entry for `dog`, once combined with the flags, the rest of the model will see different embeddings if the original text contains `Dog`, `dog's`, or `dogs`. When the model learns something new about this thing we call `dog` (like "dogs bark"), the update to the embedding matrix will affect all variants of `dog` (not to mention all variants of `bark`).

I tried two other methods of combining flags and embeddings:

- Chopping down the embedding matrix from 768 to 761, then concatenating the 7 flags. This worked but seemed awkward because it mixed flag values (-0.5 or 0.5) with learned embedding values.
- Passing the flags through a linear layer to expand them from 7 to 768, then summing them with the embeddings. This worked fine.

There is probably a good reason to prefer one of these over the other that is beyond my current level of understanding.

Extra head (for masked language modeling)

The R2D model variant for MLM has two heads, one to predict the token (no change from standard RoBERTa) and a second, similar head to predict flags. More on this in the section on training, which starts in about 2cm.

Training

For masked language modeling

I'm using the usual BERT-style MLM to mask/replace/keep 15% of tokens. For each token that is masked, I mask out all the flags. The objective is to predict the masked token *and* each of the flags. E.g. if the model thinks the masked word is `Dog's`, it must predict the token ID for `dog` with the flags `FirstCap = 1, Contraction_s = 1` and `0` for the rest.

These are calculated as two separate losses, using cross entropy for the token prediction and binary cross entropy for the flag predictions. The two losses are then added together. I tried using a λ value to amplify the flag loss (MLM loss was much larger) but it didn't improve the results.

Since the flags are quite sparse, I do per-flag weighting of the loss calculation for the flags with `torch.nn.BCEWithLogitsLoss(pos_weight=pos_weights)` based on the observed frequencies of each flag in a sampling of 10,000 documents from OpenWebText. Training is very sensitive to these weights. When I included flags that are quite rare (occurring for less than 0.1% of tokens) the multiplicative value is quite high (1,000+) and this seems to hurt training. Simply clamping the values improves results, which makes me think something is wrong with my logic here, or that there's a better way to do prediction for sparse binary values.

For reference, the occurrence rates for each flag are:

- No space before: 23%
- First cap: 18%
- All caps: 1.1%
- Plural: 3.0%
- `ed` ending: 1.7%
- `ing` ending: 1.3%
- `'s` ending: 0.9%

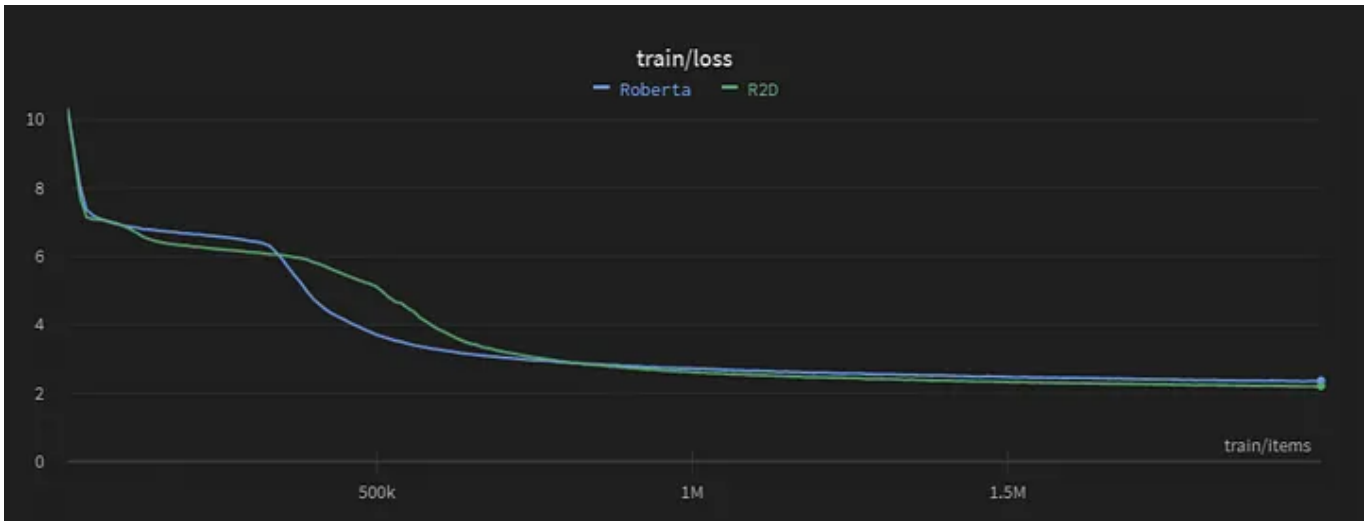
The results

Pre-training

For pre-training, I ran the experiments for 2,000,000 items (web pages) from OpenWebText. This is 0.05% (one two-thousandth) of the training of the original RoBERTa, which ran for 500k steps with a batch size of 8k.

I truncated the documents at 512 tokens rather than do something clever (like the “FULL-SENTENCES” approach described in the RoBERTa paper) since I’m primarily concerned with a comparison, not getting State-Of-The-Art results.

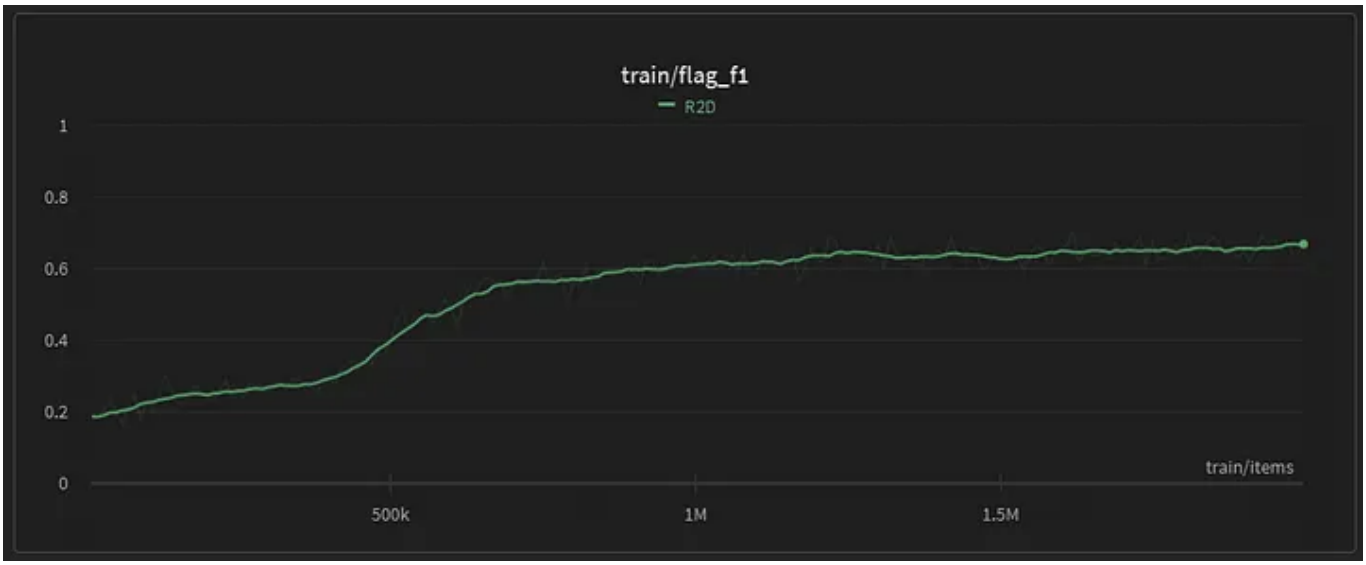
A comparison of loss looks like this, and is rather interesting…



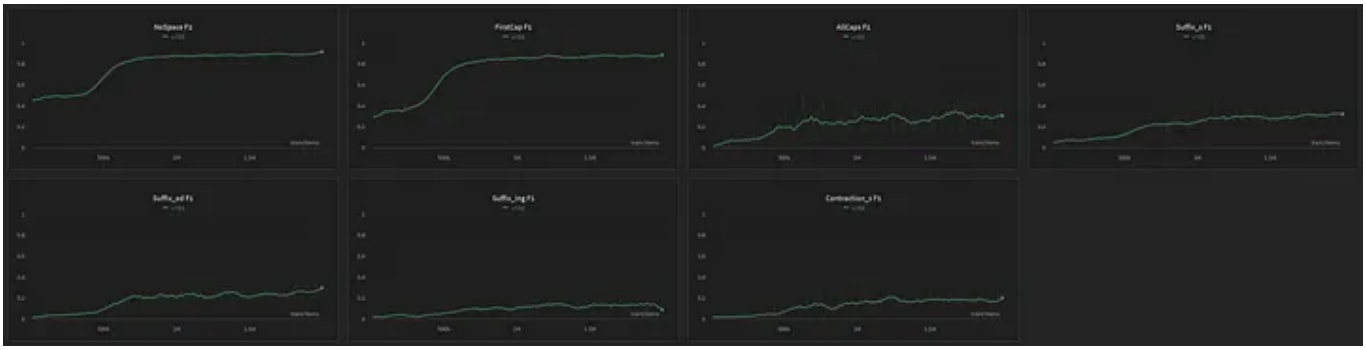
This is not a strictly apples-to-apples comparison; the R2D variant produces fewer tokens for any given item (because it has more whole words in its vocab) meaning that it's trying to predict slightly more content. (I didn't try 'whole word masking' as described in the BERT GitHub repo.)

A further disadvantage against R2D is that the loss is a sum of MLM loss and flag loss (while RoBERTa is just MLM loss). Given these differences, the fact that they're at all close to each other is interesting.

For the flags I logged the F1 score for the flag prediction objective (since most flags are zero, accuracy is not helpful). As you can see, the model gets reasonably good at predicting flags.



More telling though is the breakdown by flag. As expected, the more common the flag, the faster the model learns to predict it:



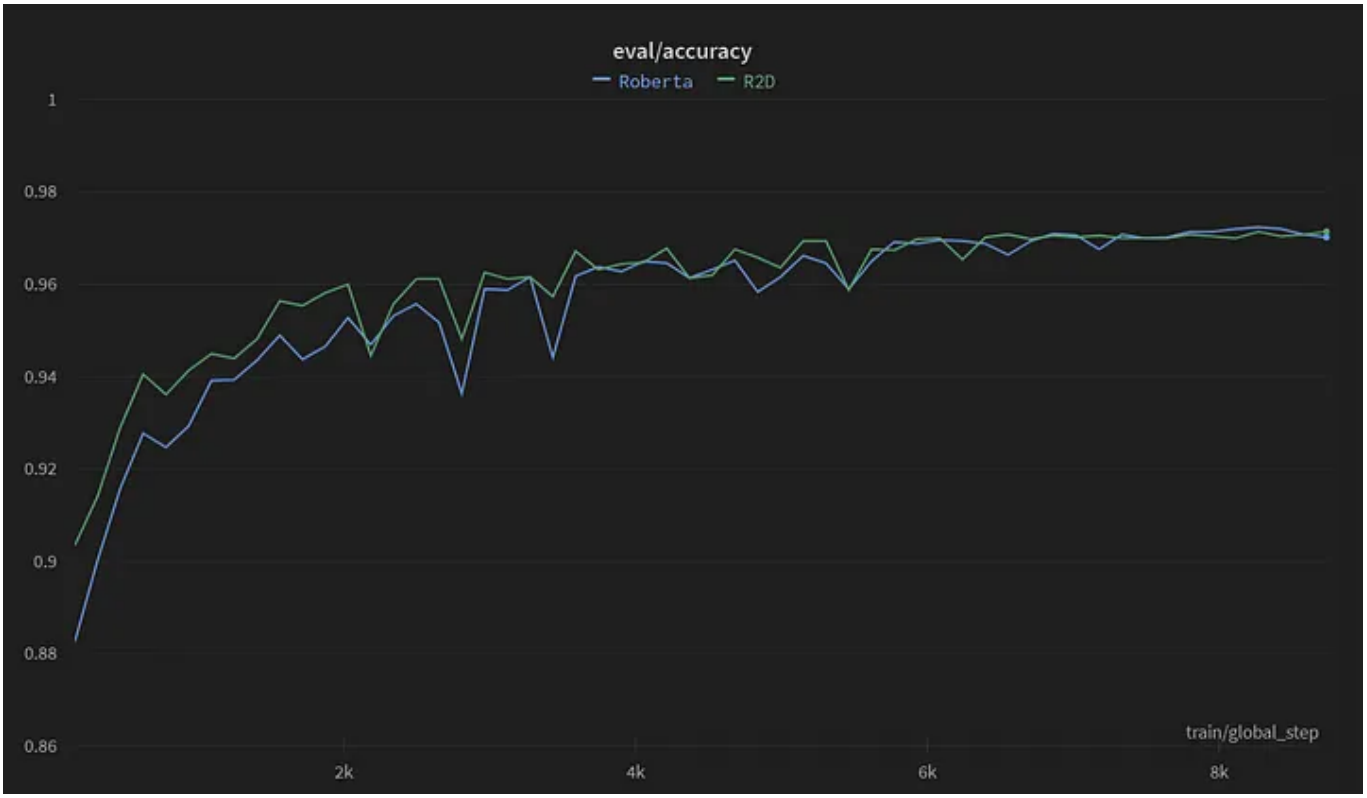
Even for the flags with lower occurrence rates, there's still a gradual increase in F1 score, so it appears that more training would help.

Now, on to downstream tasks.

Rather than the customary table of numbers, which in my view discards too much information, I'll show the training chart for each task.

Yelp polarity

The Yelp Polarity dataset involves predicting whether a given review is positive (4 or 5 stars) or negative (1 or 2 stars). (Fun fact: the original paper states that reviews of 3 or 4 stars were considered positive, but this is incorrect.)



Although R2D gets off to a good start, I'd say that by the end there's no real difference, both getting just over 97% accuracy. For reference, SOTA is 98.6% according to the Papers with Code leaderboard.

Side note: because my main goal is a comparison with a similarly-trained RoBERTa model, I haven't bothered with fine-tuning refinements like gradual unfreezing or truncating the start instead of the end (which gives better results with sentiment analysis of long reviews because disappointed diners tend to talk about how high their expectations were before eventually getting to the point that the food sucked and the staff were rude).

Next, results for a handful of tasks from the General Language Understanding Evaluation benchmark, a collection of datasets for evaluating natural language understanding systems. I'm using GLUE instead of SuperGLUE because some of the SuperGLUE tasks require referencing spans in the original text (e.g. WiC and WSC) which would require quite a bit more work with the 2D form and not tell me anything that non-super GLUE can't.

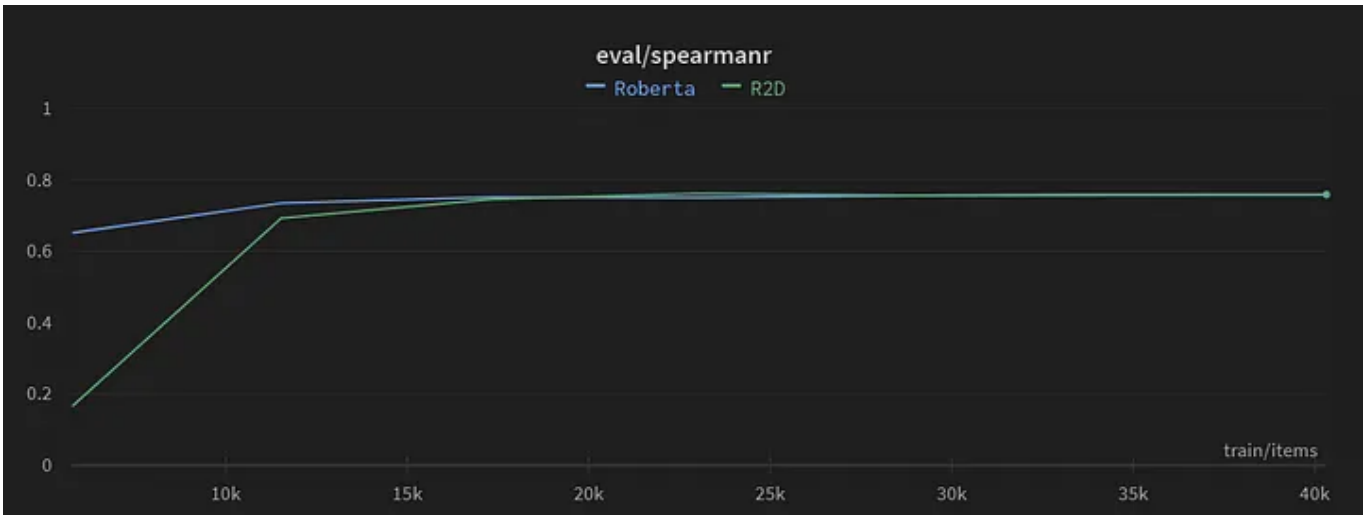
GLUE/STSB

The Semantic Textual Similarity Benchmark involves predicting how semantically similar two sequences are, on a scale of 0 to 5. (Fun fact: the GLUE paper states that these are on a scale of 1 to 5, but this is incorrect.)

For the record, I don't agree with all scores. I think that *"The man hit the other man with a stick"* and *"The man spanked the other man with a stick"* are two rather different propositions, but the dataset ranks them as 4.2.

Some of the pairs use different variants of the same root word, such as *"A woman peels a potato"* and *"A woman is peeling a potato"* which I had thought might give R2D an advantage, but these are few and far between, and not all of them use words that are in my list of deconstructions.

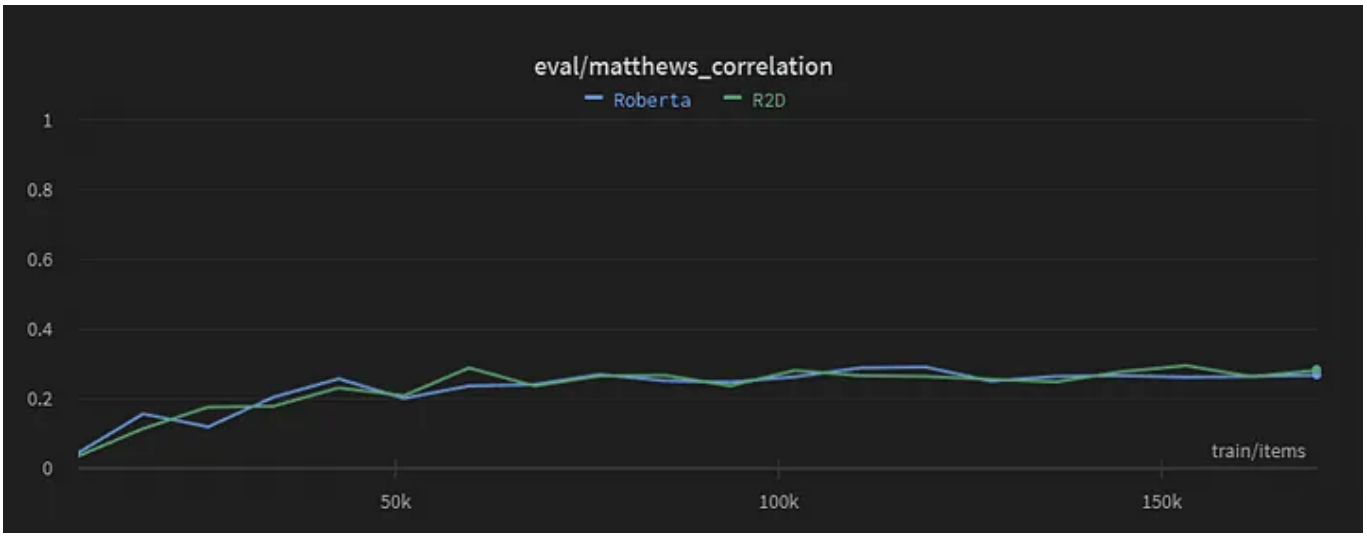
The results are that R2D and RoBERTa performed similarly.



GLUE/COLA

The Corpus of Linguistic Acceptability is one of the tougher GLUE tasks, with SOTA currently at 75.5. The content can be quite subtle, for example knowing that *"Harry coughed himself into a fit"* is 'linguistically acceptable' but *"Harry coughed himself"* is not (especially when the pre-training data is unceremoniously truncated to 512 tokens, so the model has grown up reading a lot of linguistically incomplete sentences).

Once again, the scores are just about the same (and a *long* way short of SOTA).



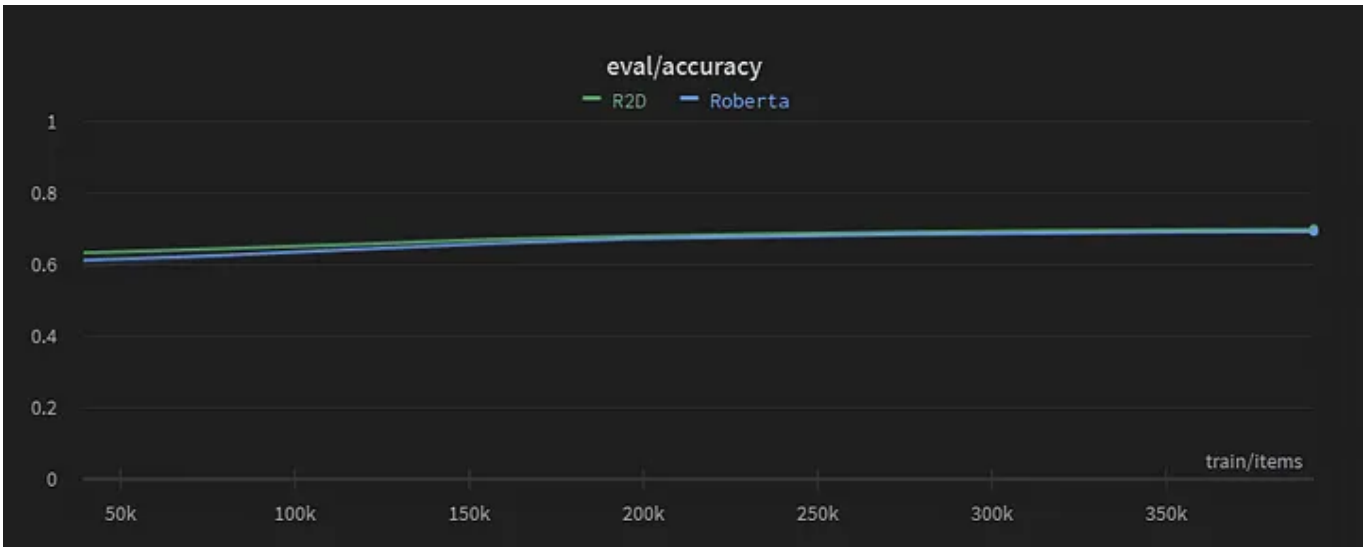
Fun fact: this GLUE task uses Matthew's correlation coefficient, which in the binary case is exactly Pearson's correlation coefficient. Is it just me, or is it odd that we have a different name for the same equation just because the data is 0's and 1's? Why not just say Pearson's? Isn't that like having a different word for multiplication on a Tuesday?

GLUE/MNLI

The Multi-Genre Natural Language Inference task asks the model to predict whether one sequence *entails* the other, *contradicts* it, or is *neutral*. An example of entailment:

- Premise: *"uh i don't know i i have mixed emotions about him uh sometimes i like him but at the same times i love to see somebody beat him"*
- Hypothesis: *"I like him for the most part, but would still enjoy seeing someone beat him"*

The results are exactly what you were expecting: very little difference.



So as you can see, at least after this small amount of training, across a smattering of downstream tasks, R2D does not impress.

The Future (for R2D)

For now, I'm parking this project. But I'll leave you with my thoughts about future avenues to explore…

More flagged words

For the main experiment, I only have 1,383 words in my replacement list and 7 flags. This could be extended, however, I think it's unlikely that this will have a dramatic effect, because the most common words have already been done (you would need to go through another 5,500 words to increase coverage from 80% to 90%).

You could do a more targeted effort, for example, just plurals, which can mostly be automated (e.g. if a word ends in s, and removing the plural gives you a word that also exists and is more frequent, you've probably got a legit plural. But there will still be exceptions like `news`, `pants`, and `glasses`).

Smarter loss weighting

The task of learning the flags means learning quite sparse binary values. I'm not certain that my current approach is the best. I don't know if it's having trouble learning `-ing` endings because it's just a difficult thing to learn, or if I'm getting something wrong in the way I've framed the objective.

More training

I had idealistic hopes of coming up with something that trained faster; that showed results in a few hours on a consumer GPU, that brought LLMs back to the people. Maybe this 2D approach *is* superior to the status quo, but only

after a few thousand TPU hours of training. Alas, I'm just a dude in his lounge room with a single lowly RTX 3090, and it's getting pretty warm in here.

ELECTRA

Some LLMs are said to be, particularly quick learners, chief among them: ELECTRA.

I spent a few days experimenting with ELECTRA early on. With ELECTRA, you don't mask the tokens, you replace them with another token and the objective is to predict *which* tokens were replaced, not what the replaced token was. I couldn't find a satisfactory way to extend that logic to binary flags with low occurrence rates. Replacing doesn't really work, and masking the flags might 'give away' which tokens have been replaced. So I switched to RoBERTa, but feel like there's more exploration to be done here.

Multilingual support

My experiment was English only, but I think the concept, in general, would map quite well across languages, as long as the flags represented concepts that exist in all (or most) languages. For example, it doesn't matter whether 'past tense' is represented by a suffix, a prefix, adding vowels in the middle of a word, or adding accents to characters. As long as the concept of past tense exists, is binary, and the root word can be identified, then the 2D flag system should be applicable.

I imagine that had this worked, I would start a GitHub repo with flags defined per language, with trusted contributors per language to field PRs.

Speaking of GitHub, I haven't open-sourced this work simply because it would require some effort (it's all nestled in a larger project), which would be

worth it if it worked, but it doesn't, so it isn't. If you want to build on this work or try it at scale, let me know and I can either just send you messy code, or if there's more interest I'll do the work and make a repo for it.

The Future (for me)

The project has served its primary purpose very well, which was for me to get intimate with large language models; I've learned an enormous amount in the time I've spent on this. (This learning objective is the reason I didn't spend a lot of time researching prior art, and the reason I'm not particularly upset that it doesn't work.)

Now I'm at a fork in my learning path:

- If someone kindly points out a stupid mistake that I've made with R2D, shining light on a potentially promising way forward, I may just follow that light.
- Otherwise, I'm off to pursue a similar learning project in the world of computer vision. Or maybe try and devise a way to do away with learning rate fiddling — it doesn't seem right that people spend so much time on trial-and-error…

Well, thanks for reading. Have a lovely day.

Data Science | Machine Learning | Deep Learning | NLP | Large Language Models

More from the list: "NLP"

Curated by Himanshu Birla

Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

· 11 min read · Sep 4, 2021

Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

· 6 min read · Sep 3, 2021

Jon Gi... in

The Word2ve

· 15 min rea

View list

Written by David Gilbertson

30K Followers · Writer for Better Programming

Follow

I like machine learning stuff.

More from David Gilbertson and Better Programming

David Gilbertson in Better Programming

87 Lesser-known Python Features

This post is for people who use Python daily, but have never actually sat down and read...

✦ · 43 min read · Jun 30

654 11

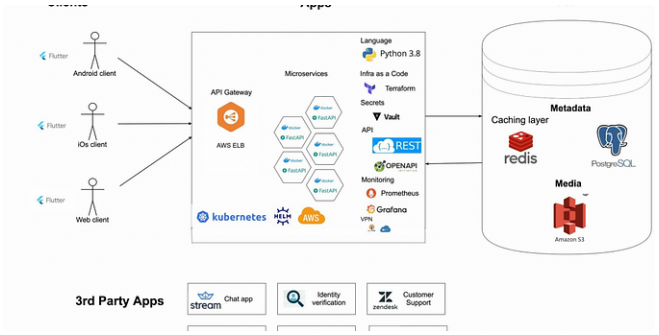
Vinita in Better Programming

How To Disagree With Someone More Powerful Than You

Disagreements can save time and energy by preventing critical mistakes and course...

✦ · 9 min read · Sep 19

1.9K 30



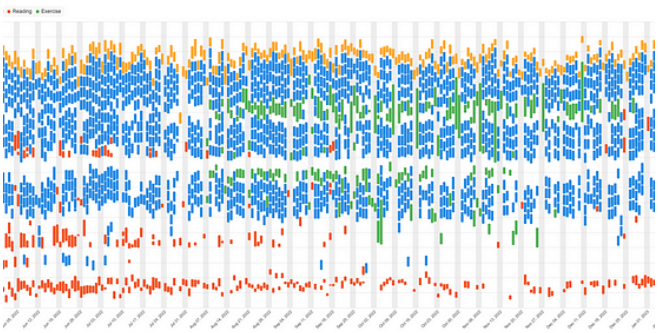
Dmitry Kruglov in Better Programming

The Architecture of a Modern Startup

Hype wave, pragmatic evidence vs the need to move fast

16 min read · Nov 7, 2022

6.1K 59



David Gilbertson in Better Programming

Can you learn machine learning in a year?

This post is for anyone considering a career move into the field of machine learning.

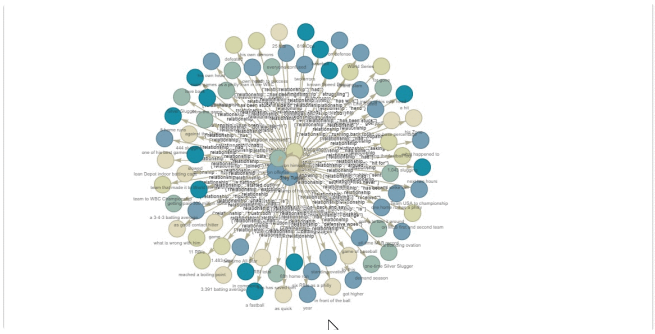
✦ · 29 min read · Jan 29

454 5

See all from David Gilbertson

See all from Better Programming

Recommended from Medium



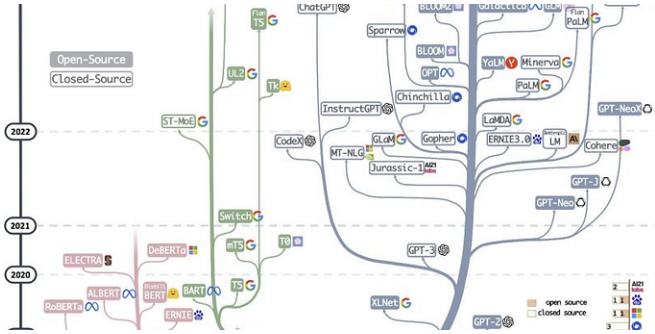
Wenqi Glantz in Better Programming

7 Query Strategies for Navigating Knowledge Graphs With...

Exploring NebulaGraph RAG Pipeline with the Philadelphia Phillies

✦ · 17 min read · 4 days ago

501 4



Haifeng Li

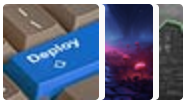
A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's...

15 min read · Sep 14

372

Lists



Predictive Modeling w/ Python

20 stories · 452 saves

Natural Language Processing

669 stories · 283 saves

Practical Guides to Machine Learning

10 stories · 519 saves

New_Reading_List

174 stories · 133 saves

David Shapiro

A Pro's Guide to Finetuning LLMs

Large language models (LLMs) like GPT-3 and Llama have shown immense promise for...

12 min read · Sep 23

283 6

Galina Alperovich in GoPenAI

The Secret Sauce behind 100K context window in LLMs: all tricks...

tldr; techniques to speed up training and inference of LLMs to use large context...

16 min read · May 16

1.4K 4

Teja Gollapudi in VMware Data & ML Blog

Efficient Instruction Fine-tuning of Flan-UL2 (20B LLM) Using LoRA...

Flan-UL2-Alpaca-LoRA

Tomas Vykruta

Understanding Causal LLM's, Masked LLM's, and Seq2Seq: A...

In the world of natural language processing (NLP), choosing the right training approach i...

7 min read · Apr 8

19

7 min read · Apr 30

20

See more recommendations