Kann eine selbstprogrammierte LM einen verständlichen, deutschen Satz erzeugen?

Albérique und Ruben L.

June 17, 2025

1 Einführung

In den letzten Jahren hat generative KI eine immer wichtigere Rolle in unserer Gesellschaft bekommen. ChatGPT, Claude, Gemini, Llama, DeepSeek, wir kennen sie alle und haben viele wahrscheinlich schon benutzt. Deshalb entschieden wir uns, in unserem Projekt, zu probieren, sie zu verstehen und dann selber ein funktionierendes Language Model (Sprachmodell) zu programmieren, ohne große Libraries zu benutzen. Funktionierend heisst hier, dass wir ein Modell haben wollen, das dazu in der Lage ist, verständliche, deutsche Sätze zu bilden. Es ist gut möglich, dass wir das nicht schaffen, deshalb ist ein realistischeres Ziel, einfach deutsche Wörter zu bilden. Neben dem Programmieren und dem Recherchieren wird es auch wichtig sein, auf unseren nicht für das Trainieren von KIs spezialisierten Laptops dieses Modell zu trainieren und dafür eine realistische Größe zu finden.

2 Tokenizing, Dataset, Embedding Layer, Vocabulary

Der erste Schritt, den wir unternahmen, war, einen Tokenizer zu programmieren. Da wir nicht noch einen Algorithmus lernen wollten, der die Tokens als Wörter oder Subwörter repräsentiert, einigten wir uns auf Zeichen-Tokenisierung (character-level tokenizing, jedes Zeichen ist also ein Token). Danach brauchten wir ein großes Dataset. Wir hatten nicht unglaublich viel Hoffnung, dass wir mit unseren bescheidenen Rechenleistungen ein sehr fähiges Modell auf die Beine bringen können, aber wir suchten dennoch ein relativ gutes Dataset, und zwar ein Dump von der ganzen Deutschen Wikipedia. Als wir mit dem Dataset zufrieden waren (es brauchte noch ein wenig Reformatieren), machten wir uns daran, ein Vokabular für das Modell zu erstellen. Alle Zeichen, die öfter vorkamen, als eine vorgegebene Zahl (Mindesthäufigkeit), wurden im Vokabular aufgenommen. Damit in Zukunft das Training schneller gehen kann, haben wir auch den gesamten Korpus, das noch aus Buchstaben bestand, in Tokens umgewandelt. Schließlich konnten wir uns der Embedding Layer widmen, unserer ersten layer. Simpel gesagt, verwandelt sie die Tokens in Vektoren, -mehr oder weniger- den ersten Neuronen! (Sozusagen die Input Layer). Diese Vektoren wurden nicht nur gemäß ihres Tokens sondern auch ihrer Position kodiert. Das bedeutet ein "l" in dritter Position hat nicht den gleichen Vektor wie ein "l" in vierter.

3 Das Modell

Nach der Embeddingschicht kommt das eigentliche Modell, das versucht, aus diesen Vektoren andere Vektoren unzuformen, die, wenn sie wieder dekodiert, für uns, nach genügend Training, verständlich sind. Es orientiert sich an der klassischen Transformerarchitektur, jedoch extrem vereinfacht und natürlich von Grund auf konstruiert (ohne Libraries, die uns groß helfen). Ein zentraler Teil unseres Modells ist das Attention-Prinzip. Vereinfacht gesagt, kann ein Neuron mit den anderen "kommunizieren" (es wird speziell mit den anderen verrechnet, damit sie sozusagen

Kontext haben). Wir haben eine extrem einfache Version davon konstruiert, doch trotzdem ist das File unser größtes.

4 Output

Damit wir das, was unser Modell da macht, auch sehen können, haben wir eine Outputschicht programmiert. Diese wandelt die Vektoren, die unser Modell am Schluss hat, in Logits um, die schon fast die Wahrscheinlichkeiten sind. Die müssen wir aber noch durch eine Softmax-Funktion jagen, um alles in den Bereich $[0, 1]$ zu bringen. Dann haben wir für jede Position einen Vektor, der angibt, wie wahrscheinlich es ist, dass ein bestimmter Buchstabe als nächstes kommt. Die Ergebnisse werden dann evaluiert (mit dem Buchstaben, der tatsächlich als nächstes kommt) und durch Backpropagation werden die Parameter aktualisiert.

5 Training

Der Training code vergleicht den vorhergesagten Token mit dem echten und gibt, mithilfe von Cross-Entropy-Loss, einen Loss an. Es berechnet anschließend den gradient vom Loss und verbessert das Modell damit, um in Zukunft bessere Vorhersagen zu bekommen. Nun aber zum interessanten Teil: Wir haben es ein paar mal trainiert und es verbessert. Eine Zeit lang verharrte die Verlustfunktion (Loss) bei etwa 4.59... und wir wussten nicht, wieso und wie man es verbessern könnte. Wir probierten, einige Werte zu verändern, wie zum Beispiel die learning rate, die Größe der Layers oder auch die Mindesthäufigkeit der Zeichen im Datenset (Ein Zeichen muss oft genug vorkommen, wir können nicht unzählige chinesische Zeichen in unser Dataset bringen). Danach fing er an, nur einen einzigen Buchstaben für mehrere Generationen immer vorherzusagen, bevor er dann nur noch zufällig " "s und "e"s spammte. Dies war zu erwarten, weil " " und "e" die wahrscheinlichsten Zeichen sind, weshalb er eine relativ hohe (höher als zufällig) Chance hatte, es richtig vorherzusagen. In dieser Phase befindet er sich normalerweise bei einem Loss von 3.5-3.9 aber sank nach noch mehr Verbesserungen, bis er anfing, neben "e" noch "h", "n", "d" und "r" zu benutzen! Das markierte einen großen Durchbruch und er konnte sogar seine ersten lesbaren Wörter kreieren, wie "er", "denen" oder "den"! Dort befand er sich auf einem Loss von ca. 2.6-3.1, was schon deutlich besser war. Leider gab es noch Probleme im Code und er ging zurück zu den "e"s und " "s.

6 Ende

Schlussendlich konnte die LM konkrete deutsche Worte bilden, aber leider haben wir etwas falschgemacht und es wird schnell "demotiviert" (geht zurück zu "e" und " ", bzw. nur noch zu " ", obwohl der Loss höher ist). Wir sind trotzdem enorm froh, dass wir es so weit geschafft haben und werden es definitiv irgendwann noch zum Laufen bringen. Damit hätten wir teilweise das konkretere Ziel erreicht, das ehrgeizigere jedoch nicht (obwohl ich davon überzeugt bin, dass es das könnte, wenn es irgendwann einmal funktioniert).

7 Quellen

Ruben hat vor geraumer Zeit in Python eine solche LM mithilfe des Tutorials [2] programmiert, aber mit sehr vielen Libraries. Davon eignete er sich jedoch ein wenig Erfahrung an, die in diesem Projekt half. Außerdem haben wir die 3Blue1Brown Videos [5] zu neuronalen Netzwerken und LLMs angeschaut. Danach haben wir ein wenig recherchiert und nach Projekten gesucht, die etwas ähnliches machen [3]. Dabei stießen wir auf einen relativ lehrreichen Huggingface-Kurs [1]. Schlussendlich ist hier noch unser Projektcode [4]

References

[1] Hugging Face. *LLM course*. War hilfreich für das Verständnis. 2024. URL: `https://huggingface.co/learn/llm-course/chapter1/1`.

[2] Andrej Karpathy. *Let's build GPT: from scratch, in code, spelled out*. YouTube. Wir haben von hier ein wenig Erfahrung mit dem Thema. 2020. URL: `https://www.youtube.com/watch?v=kCc8FmEb1nY`.

[3] Sebastian Raschka. *Github Repo mit LM in Python*. GitHub. Inspiration. 2024. URL: `https://github.com/rasbt/LLMs-from-scratch?tab=readme-ov-file`.

[4] Albérique und Ruben L. *Halb funktionierende LM in Rust (GitHub Repository)*. GitHub. Projektcode. 2025. URL: `https://www.github.com/DaCUtePotato/lm`.

[5] 3B1B (Grant Sanderson). *Neural Network Series*. Youtube. Diese Serie und ein paar andere 3b1b Videos haben uns enorm geholfen, alles zu verstehen. 2017. URL: `https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3`.