Kann eine selbstprogrammierte LM einen verständlichen, deutschen Satz erzeugen?

Albérique und Ruben L.

June 17, 2025

1 Einführung

In unserem Projekt geht es darum, ein funktionierendes LM(Language Model)-Modell zu programmieren, ohne große Libraries zu benutzen. Funkionierend 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 weniger ehrgeiziges 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 machen. Da wir nicht noch einen Algorithmus lernen wollten, der die Tokens als Wörter oder Subwörter repräsentiert, einigten wir uns auf character tokens (jedes Zeichene 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. Im Prinzip war es sehr simpel: Wir gingen durch alle Buchstaben im Dataset und die, die mehr als 1000-mal erscheinen (die also wahrscheinlich wichtig waren) wurden zusammen mit ihrem Token in ein seperates File gepackt. Damit in Zukunft das Training schneller gehen kann, haben wir auch das gesamte Dataset, 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 codiert. Das heisst ein "l" in dritter Position hat nicht den gleichen Vektor wie ein "l" in vierter.

3 Das Modell

Nach der Embeddingschicht kommt das eigenliche Modell, das versucht, aus diesen Vektoren andere Vektoren zu machen, die, wenn sie wieder decodiert, 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 erwähnenswerter 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 blieb es nur bei einem Loss von 4.59... und wir wussten nicht, wieso und wie man es verbessern könnte. TODOOOOOOOOOOOO: How did we fix it? 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 machte Sinn, 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 verbesserte 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 Quellen

Ich habe keine Zeit mehr, die Quellen anzufügen, aber wir haben eine Menge. Wenn Sie den Code in seinem jetzigen Zustand sehen wollen (er spammt "o"s bei jeder Prediction, funkioniert also noch nicht, beim Zeitpunkt des Schreibens), haben wir den Link zu unserer Github-repo hier:
www.github.com/DaCUtePotato/lm

7 Logs

1. Er hat eine Zeit lang nur 4.59... Loss gehabt und blieb 2. Dann hat er einfach nur einen Character auf's mal gespammt 3. Dann hat er nur "e"s und " " gespammt