Theory – Email Classification using Transformers

Tim Terbach

June 2025

Contents

1 Einleitung

Dieses Projekt hat das Ziel, tiefer in den Bereich der LLMs und NLP einzutauchen. Dabei werde ich erstmals Hugging Face und PyTorch verwenden, um BERT per Fine Tuning auf vordefinierte Labels zu trainieren. Somit vertiefe ich weiter meine Kenntnisse in Python, gerade im Bezug auf top-relevante Bereiche wie Fine Tuning von vortrainierten LLMs, Text-Klassifizierung und Anwendung von KI-Modellen zum Lösen praktischer Probleme. Diese Inhalte gehen weit über das im Studium erlernte Wissen hinaus, da ich jedoch das Lösen von Problemen mithilfe von KI-Anwendungen als die Zukunft des Software Engineering betrachte, gibt es wohl kaum einen besseren Zeitpunkt damit zu beginnen, als jetzt.

Fokus lege ich dabei stets auf Verständnis und Erklärung, weshalb sich diese Datei vor allem mit den theoretischen Aspekten befasst, ohne dabei zu tief in die Theorie einzutauchen. Die Balance zwischen theoretischem Wissen und praktischer Anwendung soll innerhalb dieses Projektes stets erhalten bleiben. Besonderen Wert lege ich dabei auf die Nachvollziehbarkeit und Transparenz meiner Entscheidungen bei der Implementierung und Entwicklung. Das bedeutet, dass jede Designentscheidung, von der Wahl des Modells bis zur Wahl der Labels, klar begründet und dokumentiert wird. Ziel ist es, ein Projekt zu schaffen, das sowohl technisch fundiert als auch für Außenstehende nachvollziehbar und zugänglich ist.

---

2 NLP, LLM, MLM?

In diesem Abschnitt wollen wir zunächst ein paar Begriffe einordnen, welche in diesem Projekt häufiger auftauchen werden oder generell im Kontext dieses Projektes verstanden werden sollten.

2.1 NLP

Im Kern geht es bei **Natural Language Processing** (NLP) darum, die komplexe und oft mehrdeutige Struktur unserer natürlichen Sprache in eine strukturierte und maschinenlesbare Form zu überführen, mit der Modelle Operationen durchführen können. Diese Aufgaben umfassen häufig:

1. **Textklassifizierung:** Das Zuordnen von Texten zu vordefinierten Kategorien (wie in diesem Projekt zur E-Mail-Klassifizierung).
2. **Spracherkennung:** Das Umwandeln von gesprochener Sprache in Text.
3. **Maschinelle Übersetzung:** Das Übersetzen von Texten von einer Sprache in eine andere.
4. **Sentimentanalyse:** Das Erkennen der emotionalen Haltung in einem Text.
5. **Informationsextraktion:** Das Extrahieren spezifischer Informationen aus unstrukturiertem Text.

Modelle wie **BERT, ChatGPT, Gemini oder DeepSeek** sind aktuelle Beispiele für die Fortschritte im NLP-Bereich. Sie nutzen fortschrittliche Algorithmen, um die Feinheiten der menschlichen Sprache zu erfassen und für diverse Anwendungen nutzbar zu machen. Eine große Rolle dabei spielt der Self-Attention-Mechanismus und die Transformer-Architektur, auf welche im Verlauf dieses Projektes noch weiter eingegangen wird.

2.2 LLM

Large Language Model (LLM) ist der generelle Überbegriff für genau solche Sprachmodelle, die auf einer riesigen Menge an Texten trainiert wurden, um NLP zu ermöglichen. Dabei werden durch verschiedene Trainingsstrategien die Modelle auf unterschiedliche Anwendungsfälle optimiert. Die Kernidee dabei ist jedoch im Prinzip immer dieselbe: Durch das Vorhersagen bestimmter Tokens (siehe Abschnitt *2.1*) lernt das Modell, die Semantik und den Kontext ganzer Textsequenzen zu verstehen. Dieses Verständnis ist die Kernidee, auf der LLMs beruhen. Die **umfassende Wissensbasis**, die durch das Training auf riesigen Datenmengen entsteht, ermöglicht es LLMs, komplexe Sprachaufgaben zu lösen und sich an vielfältige Anforderungen anzupassen.

2.3 MLM

Masked Language Modeling (MLM) bezieht sich auf eine Trainingsstrategie für bestimmte LLMs. Dabei wird – vereinfacht gesagt – ein Modell darauf trainiert, einzelne Tokens an zufälligen Stellen in den Trainingsdaten zu erraten".
Betrachten wir zum Beispiel den Satz *Die Sonne scheint und der Himmel ist blau"*, so würde das Modell im Training eines der Tokens durch einen `[MASK]`-Token ersetzen.
`"Die Sonne scheint und der [MASK] ist blau"`
Während des Trainings versucht das Modell nun, diesen Token vorherzusagen. Liegt es falsch, so werden die Parameter des neuronalen Netzes so angepasst, dass beim nächsten Mal mit einer erhöhten Wahrscheinlichkeit `Himmel` zurückgegeben wird – so lernt das Modell.
Ein wichtiger Unterschied zu bekannten generativen LLMs wie ChatGPT, Gemini oder DeepSeek, die häufig auf **Decoder-Architekturen** basieren, ist dabei, dass Modelle, die mittels MLM (wie **BERT**, welches eine **Encoder-Architektur** nutzt) trainiert werden, nicht nur den links stehenden Kontext verwenden, sondern auch die Tokens, die nach unserem maskierten Token folgen.
Während ChatGPT also beispielsweise während des Trainings versucht, nur mit `"Die Sonne scheint und der"` den nächsten Token (`"Himmel"`) vorherzusagen, nutzen MLM-Modelle wie BERT zusätzlich die Tokens `"ist blau"`.
Das bietet den Vorteil, dass BERT in der Lage ist, den Kontext der Tokens vor **und** nach unserem maskierten Token zu nutzen. Deswegen spricht man bei Modellen wie BERT auch von einem bidirektionalen Kontext", während Modelle wie ChatGPT lediglich einen unidirektionalen Kontext" verwenden.
Der Fokus liegt bei MLM also mehr auf dem **Verständnis einer gesamten Textsequenz**, während bei Modellen wie ChatGPT der Fokus auf dem **sequentiellen Generieren** von Tokens auf Basis bereits gesehener Tokens liegt.

---

3 Tokenizer

In diesem Abschnitt beschäftigen wir uns mit dem **Tokenizer**. Der Tokenizer ist ein zentrales Konzept jedes LLMs, das unsere natürliche Sprache in eine solche übersetzt, mit der unser Modell arbeiten kann.

3.1 Tokens

Ein LLM arbeitet mit **Tokens**; diese repräsentieren die Sprache bzw. das Alphabet des LLMs und können dabei ganz unterschiedlich aussehen. Betrachten wir zum Beispiel den Satz: *"Die Sonne scheint und der Himmel ist blau"*. So könnte hier jedes einzelne Wort ein solcher Token sein:

1. Die
2. Sonne
3. scheint
4. und
5. der
6. Himmel
7. ist
8. blau

Im nächsten Abschnitt sehen wir, dass das nicht immer der Fall sein muss. Zu Erklärungszwecken gehen wir hier aber weiterhin davon aus, dass jedes Wort als einzelner Token interpretiert wird. Wichtig ist auch, dass – wie wir in der Implementierung noch sehen werden – jedem Token ein Index zugeordnet wird. Gebe ich also als Eingabe *"Die Sonne scheint und der Himmel ist blau"*, so würde der Tokenizer diesen Satz in ein Array etwa folgender Struktur umwandeln: `[12,122,143,1543,1245,8535,245,3688]`; dabei würde nun jeder dieser Indizes für einen Token stehen:

- `12 → Die`
- `122 → Sonne`

...

Das verringert den Rechenaufwand, da mit diesen IDs schneller und effizienter gearbeitet werden kann als mit Strings. Womit genau gearbeitet wird und worauf sich diese Indizes konkret beziehen, sehen wir in einem späteren Abschnitt.

Nun stellen sich natürlich einige Fragen, wie genau diese Token generiert werden:

- *Wird einfach jedes Wort als Token gezählt?*
- *Wie viele Token gibt es?*
- *Wer definiert diese Token?*

Mit diesen Fragen beschäftigen wir uns im nächsten Abschnitt.

3.2 Der Tokenizer: Von Text zu Tokens

In diesem Abschnitt beschäftigen wir uns mit dem **Tokenizer**. Er ist ein zentrales Konzept jedes LLMs und übersetzt unsere natürliche Sprache in ein Format, mit dem das Modell arbeiten kann.

Die grundlegende Idee des Tokenizers ist es, **Texte möglichst effizient in kleinere Einheiten, sogenannte Tokens oder Subtokens, zu zerlegen**. Ziel ist es, die **Anzahl der benötigten Tokens zu minimieren**, da jeder Token einen bestimmten Ressourcenverbrauch bei der Verarbeitung durch das Modell verursacht. Eine geringere Tokenanzahl pro Text reduziert die Rechenlast und beschleunigt die Verarbeitung.

Nehmen wir an, wir würden jedes Wort als separaten Token werten – das scheint zunächst intuitiv. Daraus ergibt sich jedoch folgendes Problem: Betrachten wir allein das Wort *"spielen"*. Wir müssten nun alle Varianten dieses Wortes als einzelne Token anlegen: *"spielen"*, *"spielte"*, *"spielten"*, *"spiele"* ... und das für alle möglichen Wörter in allen möglichen Sprachen.

Effizienter wäre es, in diesem Fall `spiel` als Token zu speichern sowie `en`, `te`, `ten`, `e`. So könnten wir bereits mit den Subtokens:

`[spiel, kauf, en, te, ten, e]`

die Wörter *"spielen"*, *"spielte"*, *"spielten"*, *"spiele"*, *"kaufen"*, *"kaufte"*, *"kauften"*, *"kaufe"* bilden, ohne jedes dieser Wörter einzeln als Token speichern zu müssen.

Genau das ist die Idee bei der **Vorbereitung der Trainingsdaten**: Eine solche effiziente Zerlegung in sogenannte **Subtokens** wird vollkommen automatisch erreicht. Dafür existieren viele verschiedene Implementierungen, z. B. **Byte Pair Encoding (BPE)**.

Bei Byte Pair Encoding wird wie folgt vorgegangen:

- **Initialisierung:** Man nimmt die Trainingsdaten – z. B. den Satz *"Die Sonne scheint wie noch nie."* – und zerlegt jedes Wort in seine einzelnen Buchstaben. Ein Sondertoken wie `</w>` markiert das Wortende. Dieser Token teilt dem Tokenizer mit, dass Subtokens nicht über Wortgrenzen hinweg erstellt werden sollen. Die Häufigkeit jedes initialen Wortes wird gespeichert, z. B. `[("D", "i", "e", </w>): 1]`.
- **Iterative Zusammenführung:** Man berechnet für alle möglichen **Subtokenpaare**, wie oft diese vorkommen. Zum Beispiel könnte auffallen, dass das Paar `("i", "e")` besonders häufig auftritt. Um Ressourcen zu sparen, wird dieses Paar zu einem neuen Token `("ie")` zusammengeführt und ins Alphabet aufgenommen. Die ursprünglichen Sequenzen werden angepasst: aus `("D", "i", "e", </w>)` wird `("D", "ie", </w>)`.
- **Wiederholung:** Dieser Vorgang wird x-Mal wiederholt (wobei x eine vordefinierte Anzahl an Merges oder eine angestrebte Vokabulargröße ist). Dabei werden die häufigsten aufeinanderfolgenden Zeichen- oder Subtokenpaare zu neuen, größeren Subtokens zusammengeführt.

Die konkrete Implementierung geht natürlich weit über dieses Grundprinzip hinaus; die Grundidee sollte jedoch durch dieses Beispiel klar geworden sein: **Minimiere die Anzahl der zu bearbeitenden Tokens durch intelligentes Zusammensetzen bereits bekannter Tokens und erweitere das Alphabet um diese.**

Im Kontext von Modellen wie **BERT**, die über die **Hugging Face Transformers-Bibliothek** geladen werden, nutzen wir häufig Varianten von BPE – z. B. den **WordPiece-Tokenizer**. Dieser ist bereits mit dem spezifischen BERT-Modell vortrainiert und sorgt dafür, dass unsere Eingabetexte in das vom Modell erwartete Format übersetzt werden.

3.3 Vector Embeddings

Wir wissen nun also, was das Ziel eines LLMs ist. Zudem wissen wir, wie diese unsere Sprache in für sie verständliche Tokens umwandeln, um mit diesen zu arbeiten. Die wirklich spannenden Fragen bleiben jedoch offen:

- Wie und was lernen die Modelle?
- Wie genau verstehen sie diese Tokens?
- Wie können sie auf Basis dieser Tokens komplexe Aufgaben ausführen, wie das Generieren von Antworten, Zusammenfassungen oder Textklassifizierungen?

Hier kommen die **Vector Embeddings** ins Spiel. Jeder Token des Alphabets bekommt ein solches Vector Embedding. Diese sind – wie der Name verrät – im Prinzip erst einmal einfach nur ein Vektor, also ein Array aus Float-Werten. Der Einfachheit halber nehmen wir hier an, dass diese immer zwischen -1 und 1 liegen.

Nehmen wir als Beispiel einen Token aus unserem Beispielsatz: `‘‘Sonne’’`. Wir haben `‘‘Sonne’’` den Index `‘‘122’’` zugewiesen, zudem weisen wir diesem Index nun einen solchen Vektor zu, der wie folgt aussehen könnte: `[-0.91, 0.91, 0.43, -0.02, 0.76...]` In der Praxis bestehen diese Vektoren aus mehreren Hundert solcher Werte. Dieser Vektor repräsentiert dabei das semantische Verständnis des Modells in Bezug auf diesen Token. Wofür genau diese verschiedenen Werte stehen? Man weiß es nicht. Das ist eine große Mission aktuell: zu verstehen, wie und was genau das Modell erlernt. Man kann es sich jedoch ungefähr wie folgt vorstellen: Jeder dieser Werte könnte für eine Art Eigenschaft stehen und die Übereinstimmung dieses Tokens mit dieser Eigenschaft ausdrücken. So könnte der erste Wert `-0.91` für "Kälte" stehen (sehr geringe Übereinstimmung), der zweite Wert `0.91` eventuell für "Wärme" (sehr hohe Übereinstimmung). Der dritte Wert steht vielleicht dafür, wie groß etwas ist, usw. Was genau aber in diesen einzelnen Werten für eine Bedeutung steckt, bleibt ein Geheimnis – wir wissen nur, dass es funktioniert.

Diese Embeddings werden, gemeinsam mit anderen Parametern, auf die wir teilweise später noch eingehen, während des Trainings erlernt und angepasst. Bei jedem gescheiterten Versuch, einen Token vorherzusagen oder einen maskierten Token zu erraten, werden diese so angepasst, dass beim nächsten Versuch die Wahrscheinlichkeit für ein richtiges Ergebnis erhöht wird. Somit passt das Modell also Schritt für Schritt seine Parameter selbst an. Wie genau das wiederum passiert, ist ein Thema für sich und würde den Rahmen dieses Projekts sprengen. Wir geben uns an dieser Stelle also zunächst mit der Erkenntnis zufrieden, dass das Modell im Laufe seines Trainings die Vector Embeddings sowie andere Parameter so anpasst, dass die Vorhersage von Tokens immer wahrscheinlicher wird.

Wir haben nun nach dem Training also folgendes Mapping: **Token** (Sonne) $\mapsto$ **Index** (`122`) $\mapsto$ **Vector Embedding** (`[-0.91, 0.91, ...]`). ...

Können wir nun also einfach die Vector Embeddings von den bekannten Tokens nutzen, um einen maskierten Token zu erraten oder neue Tokens zu generieren? Leider ist das nicht der Fall – die Vector Embeddings dienen jedoch als Grundlage dafür. Um zu verstehen, wie genau diese Tokens vorhergesagt werden können, müssen wir noch etwas tiefer in die Theorie eindringen – genauer gesagt zum Herzstück unserer LLMs, dem **Self-Attention-Mechanismus**.

3.4 Self-Attention-Mechanismus

Der **Self-Attention-Mechanismus** ist dafür verantwortlich, mehrere Tokens in einen Kontext zueinander zu setzen. Mithilfe dieses Mechanismus erkennt das Modell, welche Tokens in welchem Kontext wie interpretiert und verbunden werden müssen, um die Semantik eines ganzen Textabschnitts korrekt zu erfassen.

Zurück zu unserem Beispiel: "*Die Sonne scheint und der Himmel ist blau*"

Nehmen wir an, wir wollen das Wort *blau* vorhersagen – wir setzen hier also unseren `[MASK]`-Token ein: "*Die Sonne scheint und der Himmel ist* `[MASK]`"

Im vorherigen Abschnitt haben wir uns bereits mit den Vector Embeddings beschäftigt. Wir gehen nun also davon aus, dass unser Modell bereits "versteht", was die einzelnen Tokens semantisch bedeuten. Was unser Modell jedoch noch nicht kennt, ist, wie wichtig ein Token im Kontext eines anderen Tokens ist – bzw. auf welche Tokens wir bei der Vorhersage eines weiteren Tokens besonderen Fokus legen müssen. In unserem Beispiel "*Die Sonne scheint und der Himmel ist* `[MASK]`" ist klar, dass `["Sonne", "scheint", "Himmel"]` die wohl wichtigsten Tokens in Bezug auf unseren maskierten Token `"blau"` sind. Um zu verstehen, wie das Modell diese Zusammenhänge erkennt und vor allem erlernt, müssen wir ein paar neue Vektoren einführen, ähnlich den Vector Embeddings:

- **Query**: Gibt an, welchen Kontext ein Token benötigt bzw. erwartet.
- **Key**: Gibt an, welcher Kontext von einem Token bereitgestellt wird.
- **Value**: Gibt an, welcher Wert ein Token zurückgibt, wenn Query und Key übereinstimmen.

Auch hier sollte man sich nicht zu sehr darauf konzentrieren, die exakten Werte dieser Vektoren verstehen zu wollen, sondern eher darauf, was diese in ihrer Gesamtheit repräsentieren und wie sie verwendet werden. Ähnlich wie bei den Vector Embeddings lautet die Antwort auf viele mögliche Fragen zum aktuellen Zeitpunkt wohl einfach: "Wir wissen es nicht genau."

Die Query-, Key- und Value-Vektoren werden basierend auf sogenannten **Gewichtsmatrizen** berechnet – nennen wir diese W_Q, W_K und W_V. Während des Trainings werden nicht nur die Vector Embeddings, nennen wir diese X, der einzelnen Tokens angepasst, sondern auch diese Gewichtsmatrizen W_Q, W_K und W_V. Anschließend werden dynamisch mithilfe der Vector Embeddings und der Gewichtsmatrizen die Vektoren Q (Query), K (Key) und V (Value) berechnet mit:

$$X \cdot W_Q = Q \tag{1}$$
$$X \cdot W_K = K \tag{2}$$
$$X \cdot W_V = V \tag{3}$$

Das Modell erlernt also zusätzlich im Training, wie es die Vector Embeddings am besten in Query, Key und Value umwandelt, indem W_Q, W_K und W_V angepasst werden.

Haben wir nun Q, K und V berechnet, so gilt ganz grundlegend: Die Ähnlichkeit zwischen dem Q-Vektor des aktuell betrachteten Tokens und den K-Vektoren der anderen Tokens bestimmt, welche V-Vektoren der anderen Tokens zur Vorhersage eines weiteren Tokens wie stark ins Gewicht fallen sollen. Einfach ausgedrückt: Mithilfe der Q, K und V-Vektoren – sowie ein paar weiterer Konzepte, auf die wir an dieser Stelle nicht weiter eingehen – erkennt unser LLM nun, welche Tokens mit welchen anderen im Kontext stehen, und kann somit genauere Vorhersagen treffen.

Anschaulich wird das vor allem, wenn wir unser Beispiel erneut betrachten: "*Die Sonne scheint und der Himmel ist* `[MASK]`" Vorhin hieß es vereinfacht:

> ...dass `["Sonne", "scheint", "Himmel"]` die wohl wichtigsten Tokens in Bezug auf unseren maskierten Token `"blau"` sind.

Die konkrete Vorgehensweise hängt hier jeweils vom Modell und den Trainingsmethoden ab – wir beziehen uns weiterhin auf BERT. Um diese Zusammenhänge im Detail zu verstehen, stellen wir uns vor, wie BERT das `[MASK]`-Token in unserem Satz verarbeitet: "*Die Sonne scheint und der Himmel ist* `[MASK]`". BERT beginnt damit, die **Query-Vektoren (Q)** des `[MASK]`-Tokens mit den **Key-Vektoren (K)** jedes anderen Tokens im Satz zu vergleichen. Dieser Vergleich liefert Ähnlichkeitswerte, die durch eine Softmax-Funktion in **Aufmerksamkeitsgewichte** umgewandelt werden. Diese Gewichte sind entscheidend, denn sie zeigen, wie stark das `[MASK]`-Token auf die Informationen der anderen Wörter "achtet". Beispielsweise erhält das Wort "Himmel" hier ein hohes Aufmerksamkeitsgewicht, da es stark mit dem gesuchten Token in Verbindung steht.

Gerade im Bezug auf LLMs wie BERT spricht man oft von mehreren **Attention Heads** (*Multi-Headed*). Dabei berechnet jeder dieser Heads eine eigene Menge an Q, K und V-Vektoren, die für verschiedene Beziehungen zwischen den Tokens stehen.

Ein Head könnte sich auf **syntaktische Beziehungen** konzentrieren:

- **Head 1** könnte lernen, dass "der" immer mit einem Nomen wie "Himmel" in Beziehung steht. Es erkennt also grammatikalische Abhängigkeiten.

Ein anderer Head könnte sich auf **semantische Beziehungen** konzentrieren:

- **Head 2** könnte lernen, dass "Himmel" oft mit Farben assoziiert wird und daher "blau" sehr relevant ist. Es erkennt Bedeutungszusammenhänge.

Somit "versteht" jeder Head eine andere Art von Beziehung zwischen den Tokens, sodass das Modell im Allgemeinen den Kontext der gesamten Textsequenz immer besser versteht – vor allem eben auch die Beziehungen zwischen den Tokens.

---

4 Nutzen vortrainierter LLMs

Durch das **Fine-Tuning** haben wir die Möglichkeit, vortrainierte Modelle an unseren spezifischen Anwendungsfall anzupassen. Das ist jedoch nicht die einzige Möglichkeit – eine weitere ist das **System Prompting**. Was diese beiden Begriffe bedeuten und wann welches dieser beiden Konzepte von Vorteil ist, besprechen wir in diesem Abschnitt.

4.1 System Prompting

System Prompting wird vor allem bei generativen LLMs wie ChatGPT, DeepSeek usw. verwendet. Dabei wird – vereinfacht gesagt – bei jedem Prompt (Input bzw. jeder Anfrage an das LLM) eine Art System Prompt" oder Initial Prompt" mit übergeben. In diesem Prompt wird dem Modell mitgeteilt, wie es sich verhalten soll, wie der Output aussehen soll und auf welche Aspekte besonderer Fokus gelegt werden soll. Grundsätzlich wird hier also das Verhalten und der Charakter des LLMs definiert. Basierend auf den Trainingsdaten und dem Sprachverständnis des Modells weiß dieses dadurch, wie es den Output anpassen soll, um das gewünschte Verhalten mit dem generierten Text in Einklang zu bringen.

4.1.1 Beispiele: System Prompting

Nehmen wir an, die Nutzerfrage lautet stets: "Wie ist das Wetter heute?" Die Antwort des LLMs könnte sich je nach System Prompt drastisch unterscheiden:

- **System Prompt:** "Du bist ein hilfsbereiter Wetterassistent."
  - **Mögliche Antwort:** "Heute wird es sonnig mit Temperaturen um 20 Grad Celsius."
- **System Prompt:** "Du bist ein professioneller Meteorologe. Gib detaillierte und wissenschaftlich fundierte Wetterberichte aus. Nenne Temperatur, Luftfeuchtigkeit und Windgeschwindigkeit."
  - **Mögliche Antwort:** "Für den heutigen Tag wird eine klare Himmelslage erwartet. Die Höchsttemperatur erreicht voraussichtlich 21,5 °C bei einer relativen Luftfeuchtigkeit von 65 %. Der Wind weht aus nordöstlicher Richtung mit durchschnittlich 12 km/h."
- **System Prompt:** "Du bist ein Wetter-Bot, der Antworten in maximal fünf Wörtern gibt."
  - **Mögliche Antwort:** "Sonnig, 20 Grad, leichter Wind."

An dieser Stelle möchte ich einen kleinen Exkurs zum Jailbreaking solcher generativer Modelle einschieben, da ich mir selbst an diesem Punkt folgende Frage gestellt habe:

> "Kann man dem Modell nicht einfach sagen, dass es die System Prompts umgehen oder ignorieren soll?"

4.1.2 Exkurs: Jailbreaking

Nun, **ja und nein**. Jailbreaking beschreibt in diesem Kontext das Prinzip, diese System Prompts zu umgehen, um das Modell aus seinem Käfig“ an Vorschriften zu befreien – daher der Begriff: **Jailbreaking**.

Betrachten wir folgendes Beispiel:

- Per System Prompt wird einem Modell gesagt: ”Du bist ein professioneller Meteorologe. Gib detaillierte und wissenschaftlich fundierte Wetterberichte aus. Nenne Temperatur, Luftfeuchtigkeit und Windgeschwindigkeit.”
- Nun schreibt der Nutzer: ”Du bist **kein** Meteorologe...” oder ”Ignoriere alle vorhergehenden Aufforderungen...”

Durch den zweiten Prompt wird der erste jedoch nicht überschrieben. Im Gegenteil: **Der Kontext wird einfach um diesen Prompt erweitert**. Das Modell würde also nun versuchen, sich gleichzeitig wie ein Meteorologe und wie **kein** Meteorologe zu verhalten – was zu unvorhersehbaren Ergebnissen führen kann. Zudem kann das Modell solche Versuche erkennen und direkt abblocken.

In der Vergangenheit genutzte Jailbreak-Prompts sind deshalb häufig sehr komplex, da sie eine komplett neue Umgebung für das Modell erschaffen, in der es scheinbar unbeschränkt agieren kann. Ein Beispiel hierfür ist ein älterer DAN-Jailbreak für **GPT-3.5**.

4.2 Fine-Tuning

Beim **Fine-Tuning** verfolgen wir einen anderen Ansatz: Wir nehmen ein bereits vortrainiertes Modell – in unserem Fall BERT – und trainieren es gezielt weiter, um es auf unseren spezifischen Anwendungsfall anzupassen. Vortrainierte Modelle sind in der Regel nicht auf einen bestimmten Anwendungsfall spezialisiert, sondern sollen standardmäßig eine breite Palette an Funktionen abdecken. Man kann sich diese Modelle wie eine Art Rohling“ vorstellen, der durch das Fine-Tuning immer weiter in die gewünschte Form gebracht wird.

In unserem Fall muss dabei vor allem auf drei Aspekte geachtet werden:

- Rechenaufwand des Trainings (*Computational Cost*)
- Overfitting des Modells durch die Trainingsdaten
- Underfitting des Modells durch die Trainingsdaten

In der Praxis versucht man, immer die goldene Mitte“ zwischen Overfitting und Underfitting zu treffen, um das Modell auch in realen Anwendungsszenarien zuverlässig nutzen zu können. Gehen wir nun näher auf diese Aspekte ein.

4.2.1 Computational Cost

Beim Training von LLMs werden – wie bereits erwähnt – bei jeder Iteration eine Vielzahl an Parametern angepasst. BERT verfügt über 110 Millionen solcher Parameter, die standardmäßig beim Training verändert werden. Diese alle zu trainieren ist nicht nur extrem rechenaufwendig, sondern häufig auch unnötig. Unser Ziel ist es, die Kernfähigkeiten des Modells weitgehend beizubehalten und BERT lediglich für unseren spezifischen Anwendungsfall anzupassen.

Dafür wird häufig die Methode der **Feature-Extraction** verwendet. Diese verfolgt genau den oben genannten Ansatz: Wir lassen das Basismodell (hier: BERT) größtenteils unverändert und trainieren nur diejenigen Layer, die für unseren Anwendungsfall wirklich relevant sind.

In unserem Fall wollen wir also BERTs Fähigkeiten im Bereich des Textverständnisses nutzen, um Textsequenzen bestimmten Labels zuzuordnen. Das Textverständnis selbst soll nicht weiter trainiert werden – wir gehen davon aus, dass das vortrainierte Modell hierfür bereits gut genug ist. Was wir gezielt trainieren möchten, ist der sogenannte **Classification Head**. Dieser wird – wie der Name bereits andeutet – auf das Modell aufgesetzt“ und erweitert das Basismodell um die Fähigkeit zur Klassifizierung. Der Classification Head nutzt die vom BERT-Modell generierte Repräsentation einer Textsequenz, um diese anschließend einem vordefinierten Label zuzuordnen.

In unserem Fall werden wir zusätzlich eine kleine Anzahl der im Basismodell enthaltenen Layer mittrainieren – konkret handelt es sich dabei um die sogenannten **Pooling-Layer**. Diese befinden sich meist am Ende des Modells und sind dafür verantwortlich, die hochdimensionale Repräsentation einer Textsequenz in eine niedrigdimensionalere Vektorform zu überführen. Dies vereinfacht spätere Berechnungen und liefert eine komprimierte Repräsentation der gesamten Eingabesequenz. Genau diese finale Repräsentation dient anschließend als Input für den Classification Head.

Durch das gezielte Anpassen der Pooling-Layer und des Classification Heads optimieren wir, wie das Modell die Informationen für unseren spezifischen Anwendungsfall zusammenfasst – ohne die komplexen inneren Transformer-Layer des Basismodells von Grund auf neu trainieren zu müssen.

BERT eignet sich besonders gut für die Zusammenfassung ganzer Textsequenzen in eine kompakte Repräsentation. Dies ist unter anderem der Trainingsmethode **Next Sentence Prediction** zu verdanken. Ohne an dieser Stelle zu sehr ins Detail zu gehen: BERT wird während des Pretrainings darauf trainiert, basierend auf einem speziellen Token `[CLS]`, der die kontextuelle Repräsentation der gesamten Eingabesequenz speichert, den nachfolgenden Satz vorherzusagen. Diesen `[CLS]`-Token nutzen wir auch als Input für unseren Classification Head. Im Training lernt BERT also, eine vollständige Textsequenz effizient in genau diesen Token zu komprimieren“ – was exakt unserem Anwendungsfall entspricht.

4.2.2 Overfitting

Beim **Overfitting** passt sich das Modell zu stark an die spezifischen Muster der Trainingsdaten an. Es beginnt, nicht nur relevante, sondern auch zufällige oder unbeabsichtigte Zusammenhänge zu lernen“, die in den Trainingsdaten enthalten sind.

Ein Beispiel: Angenommen, unsere Trainingsdaten bestehen aus E-Mails mit jeweils zugehörigem Label. Das Modell soll diese E-Mails korrekt klassifizieren. Beginnen nun alle E-Mails mit dem Label Bewerbung“ mit dem Satz Sehr geehrte Damen und Herren, mein Name ist. . . “, könnte das Modell lernen, dass jede E-Mail, die so beginnt, automatisch zum Label Bewerbung“ gehört – obwohl es eigentlich auf den Gesamtinhalt und Kontext ankommen sollte.

In solchen Fällen kann sich das Modell an bestimmte Wortwahlen, Namen oder Satzstrukturen klammern“, weil es in den Trainingsdaten funktioniert – nicht aber in der Realität. Overfitting äußert sich häufig darin, dass das Modell auf dem Trainingsdatensatz sehr gute Ergebnisse erzielt, aber bei unbekannten Beispielen, die leicht vom Trainingsmuster abweichen, versagt.

Vereinfacht gesagt: Ein überangepasstes Modell lernt zu sehr die Besonderheiten der Trainingsdaten und verliert seine Fähigkeit zur **Generalisierung** auf neue, unbekannte Daten – es wird also starr“.

Overfitting kann typischerweise durch folgende Ursachen entstehen:

- Das Modell ist zu komplex (z. B. zu viele Layer oder Parameter im Verhältnis zur Datenmenge).
- Zu lange Trainingsdauer.
- Zu kleiner Trainingsdatensatz (zu wenig Variation).
- Fehlerhafte oder **verrauschte** Trainingsdaten.

Verrauscht bedeutet hier: Der Trainingsdatensatz enthält irrelevante oder störende Informationen, auf die sich das Modell fälschlicherweise konzentrieren könnte.

4.2.3 Underfitting

Während Overfitting beim Fine-Tuning vortrainierter Modelle häufiger auftritt – da diese bereits über ein starkes Sprachverständnis verfügen –, kann in manchen Fällen auch **Underfitting** problematisch sein.

Underfitting bedeutet, dass das Modell nicht in der Lage ist, die zugrunde liegenden Muster in den Daten zu erkennen. Das kann z. B. passieren, wenn der für das Fine-Tuning verwendete Datensatz zu klein ist oder die gewählte Trainingskonfiguration (etwa eine zu geringe Lernrate) das Modell daran hindert, neue Zusammenhänge zu lernen.

In unserem Anwendungsfall erwarten wir allerdings kein nennenswertes Underfitting. Deshalb soll dieser Aspekt an dieser Stelle nicht weiter vertieft werden.

4.3 System Prompting vs. Fine-Tuning

Abschließend stellt sich die Frage, wann welches der beiden Konzepte sinnvoll angewendet werden sollte.

System Prompting ist in der Regel immer dann empfehlenswert, wenn ein Modell vielseitig genutzt wird und sich die Anwendungsfälle häufig ändern oder generell sehr dynamisch sind. Dabei geht es darum, das Modell *temporär* für eine bestimmte Aufgabe oder einen spezifischen Stil zu instruieren, ohne seine Kernkompetenzen dauerhaft zu verändern.

Typische Anwendungsbeispiele für System Prompting:

- Kundenservice-Chatbots mit wechselnden Anforderungen
- Inhaltsgenerierung für Social Media
- Sprachübersetzung oder Textgenerierung in spezifischen Stilen

Fine-Tuning wird hingegen bevorzugt eingesetzt, wenn – wie in diesem Projekt – der Anwendungsbereich des Modells sehr statisch ist und somit regelmäßig die gleiche Aufgabe ausgeführt werden soll. Hier steht die *dauerhafte* Optimierung des Modells für eine spezifische Aufgabe im Vordergrund, mit dem Ziel, eine möglichst hohe Genauigkeit und Qualität zu erzielen.

Typische Anwendungsbeispiele für Fine-Tuning:

- Textklassifizierung mit festen Labels
- Sentiment-Analyse von Textsequenzen
- Medizinische oder juristische Textauswertung

---

5 Ergebnisse

In diesem Abschnitt befassen wir uns mit den Metriken, die wir zur Auswertung und Beurteilung der Leistungsfähigkeit unseres Modells nach dem Fine-Tuning nutzen können. Hierbei unterscheiden wir generell zwei Kategorien: Die erste Kategorie, die **allgemeinen Metriken**, bewertet die Genauigkeit und Zuverlässigkeit unseres Modells über alle Labels hinweg. Die zweite Kategorie, die **label-spezifischen Metriken**, fokussiert sich auf detailliertere, auf einzelne Labels bezogene Werte.

5.1 Allgemeine Metriken

Wie bereits erwähnt, wird hierbei die Leistung und Genauigkeit unseres Modells in Bezug auf alle Labels betrachtet; es findet also keine Differenzierung zwischen den einzelnen Labels statt. Für dieses Projekt haben wir uns dabei für folgende Metriken entschieden, da diese, ohne zu tief in die Theorie eintauchen zu müssen, bereits einen guten Einblick in die Qualität unseres Modells bieten und unerwünschte Effekte wie Overfitting erkennbar machen.

5.1.1 Trainings-Loss

Der **Trainings-Loss** misst, wie gut das Modell auf den Trainingsdaten performt. Über die Trainingsiterationen ist ein sinkender Wert erwünscht. Dieser Wert wird verwendet, um den Lernfortschritt zu überwachen; sollte er nicht sinken, scheint das Modell aus den Trainingsdaten nicht zu lernen.

5.1.2 Validierungs-Loss

Der **Validierungs-Loss** misst die Fehler des Modells auf einem separaten Satz von Validierungsdaten, die das Modell während des Trainings nicht gesehen hat. Ein niedriger Wert zeigt an, dass das Modell die aus den Trainingsdaten gelernten Muster auch gut auf unbekannte, neue Daten anwenden kann. Diese Metrik ist entscheidend zur Erkennung von Overfitting: Sinkt beispielsweise der Trainings-Loss (Modell performt gut auf Trainingsdaten) und gleichzeitig steigt der Validierungs-Loss (Modell performt schlecht auf neuen, unbekannten Daten), so ist dies ein starkes Indiz für Overfitting.

5.1.3 Accuracy Score

Der **Accuracy Score** (Genauigkeitswert) ist eine einfache, intuitive Einschätzung der Gesamtperformance des Modells. Er gibt den Anteil der korrekt klassifizierten Validierungsdaten wieder.

5.1.4 AUC Score

Der **AUC Score** (Area Under the Curve) bewertet die Fähigkeit des Modells, zwischen verschiedenen Labels zu unterscheiden. Ein höherer Wert (näher an $1,0$) bedeutet, dass das Modell gut zwischen den verschiedenen Klassifizierungs-Labels unterscheiden kann. Der AUC Score bietet somit eine robustere Bewertung der Leistung unseres Modells als die reine Accuracy, insbesondere bei unausgeglichenen Label-Verteilungen.

Beispiel: Nehmen wir an, wir haben zwei Labels und das Modell würde jeden Text der Validierungsdaten Label 1 zuordnen. Bei einer Verteilung von **70% Label 1** und **30% Label 2** in unseren Validierungsdaten hätten wir somit eine **Accuracy von 70%**. Der Wert der Accuracy wäre in diesem Fall jedoch etwas irreführend, da das Modell tatsächlich sehr schlecht in seiner Aufgabe ist, die Texte den entsprechenden Labels zuzuordnen. Hier würde der AUC Score eine bessere Einschätzung der Performance unseres Modells bieten, da dieser in diesem Beispiel sehr niedrig wäre.

5.2 Label-spezifische Metriken

Hier schauen wir uns nun die Metriken an, welche label-spezifische Aussagen über Genauigkeit und Leistung treffen können. Auch hier wurden die Metriken dem Projekt entsprechend gewählt und sollen vor allem die Stärken und Schwächen unseres trainierten Modells aufzeigen.

5.2.1 Precision

Die **Precision** (Präzision) zeigt für jedes Label, wie viele der vom Modell diesem Label zugeordneten Daten tatsächlich diesem Label zugehörig sind. Eine höhere Precision bedeutet also, dass das Modell bei der Zuordnung zu diesem Label selten falsch liegt. Das ist besonders von Bedeutung, wenn beispielsweise die korrekte Zuordnung zu einer bestimmten Kategorie von hoher Priorität ist (d. h., falsch-positive Klassifikationen vermieden werden sollen).

5.2.2 Recall

Der **Recall** (Vollständigkeit) gibt an, wie viele der zu einem Label gehörigen Validierungsdaten vom Modell tatsächlich als solche erkannt und korrekt zugeordnet wurden. Auch hier sollte man größeren Fokus darauf legen, wenn es von großer Bedeutung ist, eine bestimmte Datengruppe vollständig zu identifizieren und keine relevanten Fälle zu übersehen (d. h., falsch-negative Klassifikationen vermieden werden sollen).

5.2.3 F1-Score

Wie bereits erwähnt, gehen **Precision** und **Recall** meist Hand in Hand und stellen in ihrer Gesamtheit eine wichtige Metrik dar. Um diese also zusammenzufassen und aus Precision und Recall eine ausgeglichene Metrik zu schaffen, nutzt man häufig den **F1-Score**. Dieser ist besonders aussagekräftig, wenn ein Gleichgewicht zwischen der Vermeidung von Fehlalarmen (hohe Precision) und dem Identifizieren aller relevanten Datensätze (hoher Recall) angestrebt wird.

5.2.4 Konfusionsmatrix

Die **Konfusionsmatrix** ist eine Tabelle, welche die Anzahl der richtig und falsch klassifizierten Daten für jedes Label darstellt. Mit einer solchen Konfusionsmatrix lassen sich besonders gut die Stärken und Schwächen des Modells aufzeigen, da zu erkennen ist, welche Labels verlässlich richtig zugeordnet werden und welche eventuell häufig miteinander verwechselt werden. Dadurch können gegebenenfalls Trainingsdaten erweitert oder verändert werden.