ML PROJEKT

Marc Engelmann
Marius Funk
Markus Budeus

Encoding

Wir haben mehrere Modelle ausprobieren wollen, unter anderem einen Binary Encoder, einen Auto Encoder und einen OneHot Encoder. Da leider bis auf den OneHot Encoder alle Encoder nicht besonders gute Performance hatten, haben wir uns für diesen entschieden, was sehr gut geklappt hat.

Modelle

Unser ursprünglicher Plan war es, 3 Modelle zu entwickeln um diese dann mit einem Mehrheitsvoting noch weiter zu optimieren. Also immer abstimmen lassen und die häufigere Entscheidung (Edge oder kein Edge) gewinnt. Aufgrund von Problemen bei einem dieser Ansätze konnten wir das am Ende leider nicht so umsetzen, auch wenn alle 3 Ansätze noch hier bestehen:

XGBoost

Ursprünglich sollte ein Ansatz aus einem Graph-Neural-Network bestehen. Nachdem klar geworden ist, dass GNNs eher für viele Daten, die von einem großen Graphen repräsentiert werden können, geeignet sind, kam die Idee auf, die gegebenen Daten nicht als Graphstruktur, sondern als tabellarische Daten zu betrachten. Da Decision Trees, genauer Random Forrests, im Rahmen von XGBoost auf tabellarischen Daten viele andere Modelle schlagen, wurde das unser nächster Ansatz.

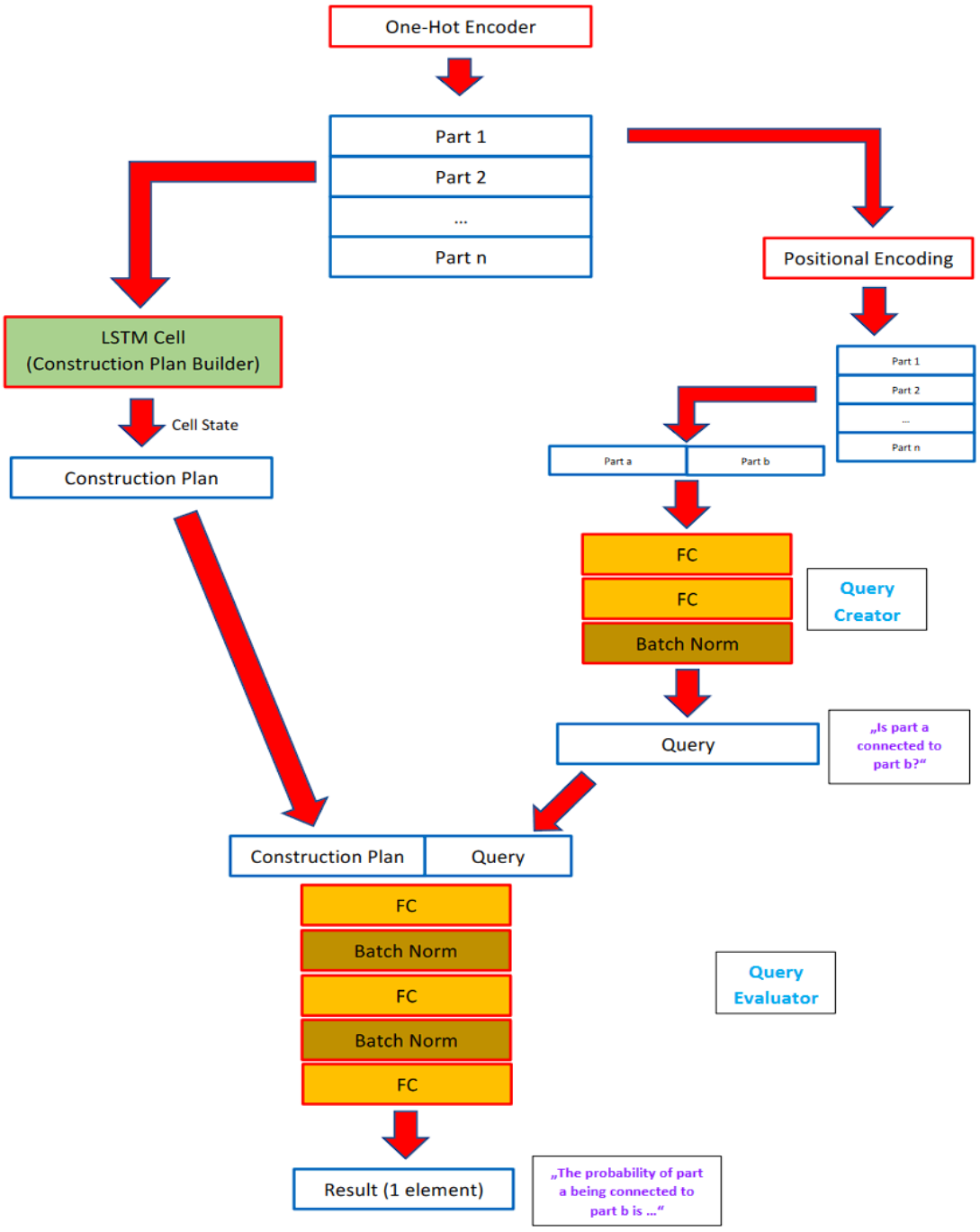
Jedes Part eines Graphen p wird durch seine Family ID repräsentiert, da diese für ähnliche Parts einen ähnlichen (gleichen) Wert besitzt. Da die Family ID bereits Ähnlichkeit beschreibt, wurde auf weitere Embeddings verzichtet.

Für einen Graphen g werden alle Paare von Parts (p1,p2) betrachtet und pro Paar ein Sample erstellt. Dieses Sample hat die Einträge:
p1_family ; p2_family ; n_parts_in_graph ; n_parts_of_fam1 ; n_parts_of_fam2 ; ... ;
Wobei *n_parts_in_graph* die Anzahl der Teile des Graphen beschreibt. Die folgenden Felder beinhalten die Anzahl der Teile mit entsprechender Family ID in dem Graphen. Die Labels sind lediglich True/False Werte für jedes Part-Paar, entsprechend, ob sie mit einer Kante verbunden sind, oder nicht.

Der XGBClassifier erstellt einen Random Forest aus 1500 Bäumen mit maximaler Tiefe 3, der bestimmt, ob ein Paar von Parts mit einer Kante verbunden sein soll, oder nicht. Wie unten beschrieben, wurde dann noch der resultierende vorhergesagte Kantenvektor optimiert. Trainiert mit 80% der Trainingsgraphen wurde eine Accuracy von über 95% auf den restlichen 10% der Instanzen erreicht, nachdem 10% der Daten zum manuellen Tunen der Hyperparamter (Anzahl Trees, Baumtiefe, etc.) genutzt wurden. Das beste Modell (*model_1500_estimators*) wird in *evaluation.py* geladen.

LSTM

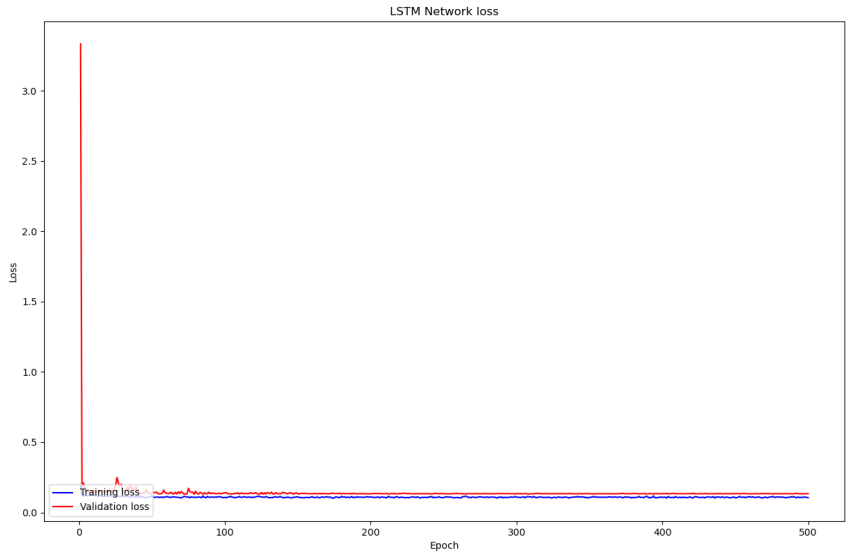
Der LSTM-Ansatz basierte auf der Grundidee, alle Bauteile nacheinander in eine LSTM Cell zu geben und danach deren Cell State als Bauplan zu interpretieren. Auf diesen Bauplan können nun Anfragen ausgeführt werden, um zu ermitteln, mit welcher Wahrscheinlichkeit zwei bestimmte Teile miteinander verbunden sind. So kann das Netz beliebig große Graphen verarbeiten. Der Aufbau des Modells ist wie folgt:



Leider scheiterte dieser Ansatz komplett. Keine Version des Modells schaffte es, mit Training eine bessere Edge Accuracy zu erreichen als ohne Training. Folgende Dinge wurden (meist auch in Kombination) ohne Erfolg ausprobiert:

- Lange Trainingszeiten (500 Epochen!)
- He Initialization aller MLPs, die mit ReLU arbeiten
- Einsatz von Leaky ReLU anstelle von ReLU
- Bauteile auch mit Positional Encoding in das LSTM geben
- Einsatz von Adam statt SGD
- Versuche mit Cross-Entropy und Binary Cross-Entropy, letztendlich sogar eine eigene Loss-Funktion auf Basis des BCEWithLogitsLoss, die False Negatives 25-fach gewichtet
- Maskierung des Paddings in der Loss-Funktion
- Einfügen von Batch Normalization
- Verschiedene Größen der Hidden Layers
- Learning Rate Scheduling
- Beobachtung, ob Gewichte explodieren (war nicht der Fall)
- Einsatz des gesamten Datensatzes als Versuch, ein Overfitting zu provozieren.

Fast immer war das Ergebnis, dass der Loss in den ersten 3 Epochen schnell abgenommen hat, danach sich aber keinerlei Verbesserung eingestellt hat. Siehe als Beispiel der Versuch mit 500 Epochen Training:



Kein trainiertes Modell erreichte eine Edge Accuracy über 72%. Allerdings wird auch von einem untrainierten Modell diese Edge Accuracy erreicht, da der Optimizer alleine dies erreichen kann. Ohne Optimizer haben die trainierten Modelle keine zusammenhängenden Graphen geliefert.

Da dieser Ansatz keine guten Ergebnisse geliefert hat, enthält unsere Abgabe kein vortrainiertes Modell dieses Typs. (Zugegebenermaßen liegt das auch mit daran, dass es zu groß ist, um von Git akzeptiert zu werden.)

FFNN

Der vermutlich simpleste Ansatz, aber auch der schlussendlich performanteste. Mit ca. 96-97% Genauigkeit sticht er die anderen Ansätze aus. Genau nach Occam's Razor sollte man wohl ein Problem nicht überanalysieren. Hier werden einfach alle Bauteile paarweise OneHot encodiert und dann durch 4 FC-Layers gejagt.

Es wird dabei jeder Graph als Batch der Größe ∑2 - n-1 (Für jede mögliche Kante, siehe "Edge Vector") mit einem 2*Encoding Size großen Tensor für beide Bauteile der Kante.

Als Lossfunktion sind wir letzten Endes auf Binary Cross Entropy gelandet, das funktionierte am Besten. Für den Optimizer nahmen wir den Gradient Descent Ansatz (SGD), der in der Vorlesung so überzeugend gezeigt wurde. Zum Initiieren wurde He Initiierung verwendet, um das Training zu beschleunigen. ReLU wurde zwischen jedes Layer gesetzt, um die negativen Werte zu entfernen.

Trainiert wurde es für 1-50 Epochen, nach 2 haben wir aber kaum noch Verbesserungen festgestellt. Dennoch wurde hier nie overfitted nach dem Trainingsset, wenn man den Tests glauben darf. Demnach haben wir es 50 Epochen laufen lassen, um alles an Performance rausholen zu können.

Sonstiges

Edge Vector

Im Programmcode kommt ab und zu der Begriff des “Edge Vector” vor. So nennen wir eine kompaktere Darstellung der Adjazenzmatrix, die sich zunutze macht, dass wir nur mit ungerichteten und antireflexiven Graphen arbeiten.
Die Adjazenzmatrix eines solchen Graphen hat die Form

0 A B C
A 0 D E
B D 0 F
C E F 0

Der dazugehörige Edge Vector ist **[A, B, C, D, E, F]**.

Optimizer

Der Optimizer ist ein rein algorithmisches Hilfswerkzeug. Es macht aus dem Edge Vector mit den entsprechenden Kantenwahrscheinlichkeiten, wie sie von einem Modell vorhergesagt wurden, einen azyklischen, zusammenhängenden Graphen.
Dies wird erreicht, indem der Optimizer einen *Minimum Spanning Tree* auf den Knoten berechnet, wobei die Kanten umgekehrt zu deren vorhergesagter Wahrscheinlichkeit gewichtet werden. Anders gesagt: Als wahrscheinlicher vorhergesagte Kanten werden bevorzugt für den Minimum Spanning Tree verwendet.

Andere Ansätze

Wie oben erwähnt, sollte ursprünglich ein GNN implementiert werden. Die Idee wurde jedoch schnell verworfen. Ein anderer Ansatz, der frühzeitig aufgegeben wurde, war ein KNN Ansatz. Da andere Ansätze schon zu dem Zeitpunkt deutlich mehr Potenzial zeigten, wurde der KNN Ansatz schließlich nicht weiter verfolgt.