Szachy bez przeszukiwania: Uczenie maszynowe z użyciem transformera w kontekście szachów i warcab

Kacper Mikołajuk 198254,
Natalia Dembkowska 198152,
Natalia Sekula 197913,
Olga Rodziewicz 198421,
Patryk Lewandowski 197891
Politechnika Gdańska

Abstract

Celem projektu było zbadanie możliwości zastosowania architektury transformera do nauczenia modelu gry w szachy oraz warcaby. W przeciwieństwie do tradycyjnych silników szachowych bazujących na algorytmicznym przeszukiwaniu kolejnych możliwości, w celu znalezienia najlepszego ruchu, trenujemy X - parametrowy transformer poprzez przekazanie X ocenionych plansz dla modelu szachowego oraz x plansz dla modelu warcabowego.

POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Spis treści

1 Wprowadzenie

Transformer to architektura sieci neuronowej zaproponowana w pracy „Attention Is All You Need" (Vaswani i in., 2017). W odróżnieniu od klasycznych rekurencyjnych sieci neuronowych (RNN) czy sieci konwolucyjnych (CNN), transformer opiera się w całości na mechanizmie *self-attention*, co pozwala mu jednocześnie przetwarzać wszystkie pozycje w sekwencji wejściowej. Kluczowe elementy transformera to:

- **Mechanizm self-attention:** Każdy token (np. ruch w partii szachów czy warcab) jest reprezentowany jako wektor, a następnie dla każdej pary tokenów obliczana jest waga (uwaga) wskazująca, jak mocno jeden token powinien wpływać na reprezentację drugiego. Dzięki temu model może wychwytywać zależności między dowolnymi elementami sekwencji, niezależnie od ich odległości pozycyjnej.

- **Embedding pozycyjne:** Ponieważ transformer nie przetwarza sekwencji w sposób rekurencyjny, konieczne jest zakodowanie informacji o kolejności tokenów. Wykorzystuje się do tego wektory pozycjonowania (ang. *positional encodings*), które dodawane są do reprezentacji każdego tokena. Dzięki temu model rozumie, które ruchy występują wcześniej, a które później.

- **Warstwy feed-forward i warstwy normalizacji:** Po obliczeniu uwag dla każdej pozycji następuje przejście przez warstwę liniową z nieliniową funkcją aktywacji (np. ReLU), a następnie warstwa normalizacji (Layer Normalization). Całość tworzy moduł, który można wielokrotnie powielać (tzw. bloki transformera).

- **Maskowanie (w modelach autoregresywnych):** Aby model ucząc się generowania sekwencji nie „zobaczył" przyszłych tokenów, stosuje się tzw. *causal mask* — maskę zabezpieczającą, która uniemożliwia uwzględnianie wagi (uwagi) od pozycji dalszych niż obecna.

W praktyce architektura transformera dzieli się na dwie główne części:

1. **Encoder:** W wielu zadaniach, np. tłumaczeniach maszynowych, stosuje się strukturę encoder–decoder. Część enkodera przetwarza całą sekwencję wejściową (np. zdanie w języku źródłowym), tworząc wektory reprezentujące znaczenie każdego tokena z uwzględnieniem kontekstu całego zdania.

2. **Decoder:** W części dekodera model generuje sekwencję wyjściową (np. przetłumaczone zdanie), krok po kroku, korzystając z informacji z enkodera i dotychczas wygenerowanych tokenów. Każdy dekoderowy blok transformera stosuje najpierw mechanizm self-attention (z maską), a następnie attention nad wyjściem enkodera.

W kontekście naszego projektu, w którym celem jest wytrenowanie transformera generatywnego do przewidywania kolejnych ruchów w partii (szachowej lub warcabowej), możemy skupić się na tzw. *transformerze decoder-only* (podobnie jak modele z rodziny GPT). Taki model:

- Przyjmuje na wejściu sekwencję dotychczasowych ruchów (zakodowanych jako wektory tokenów).

- A dzięki mechanizmowi maskowanego self-attention uczy się generować kolejny token (ruch) bazując na całości dotychczasowej sekwencji.

2 Implementacja

2.1 Zbiór danych

Dane użyte w projekcie zostały przygotowane w następujący sposób:

1. **Pozyskanie surowych danych:**
   W przypadku szachów, dane został pobrane z serwisu LiChess w formacie PGN, które przekształcamy na notację FEN.

   W przypadku warcab, dane pochodzą z LiDraughts w formacie X. Dodatkowo dla warcab została zaimplementowana opcja nauki w trakcie gry Model vs Model.

2. **Ocena pozycji:**

   - *Szachy:* Dla każdej pozycji (zapisanej w formacie FEN) uruchamiony zostaje silnik Stockfish za pomocą biblioteki `python-stockfish`, który dla danej planszy zwraca najlepszy ruch.
   - *Warcaby:* Aby uzyskać najlepszy ruch, użyty został wrapper pydraugths, korzystający z zewnętrznego silnika Scan Engine dostosowanego do warcab na planszy 10×10. Skrypt w Pythonie przekazywał pozycję w postaci odpowiedniego formatu (analogicznego do FEN dla warcab), a silnik zwracał najlepszy ruch w notacji.

3. **Baza danych:**
   Do składowania plansz użyta została baza danych SQLite. Przekształcone dane na odpowiednią notacje wraz z najlepszym ruchem uzyskanym przez silnik szachowy/warcabowy zostają przechowane w bazie danych. Następnie jest ona źródłem danych dla uczonego modelu transformera.

2.2 Model

Implementacja modelu oraz procesu treningu została zrealizowana w języku Python, z wykorzystaniem bibliotek `torch` (PyTorch) i `pytorch-lightning`. Kluczowe etapy to: tokenizacja, konstrukcja architektury transformera oraz algorytm treningowy.

Tokenizacja danych Aby model transformera mógł operować na danych, każdą pozycję oraz odpowiadający jej ruch należało zamienić na sekwencję liczb (tokenów). W pierwszym kroku dzieliliśmy tekstową notację FEN na fragmenty odpowiadające poszczególnym rzędom szachownicy (lub układowi warcabnicy), a następnie każdy znak (litery figur, cyfry określające puste pola, symbole rozdzielające) traktowaliśmy jako osobny token. Dodatkowo w FEN-ie zawarte są informacje o aktywnym kolorze, prawach do roszady, kwadracie *en passant* oraz licznikach półruchów i ruchów; wszystkie te elementy kodowaliśmy jako odrębne tokeny liczbowo odpowiadające ich możliwym wartościom. W efekcie, dla każdej pozycji

uzyskiwaliśmy sekwencję o długości w praktyce oscylującej wokół 70–80 tokenów, co wliczało zarówno układ bierek, jak i pozostałe parametry FEN-a.

Ruch w notacji UCI (np. `e2e4`) lub analogiczny w warcabach (np. `b6c7e5`) również dzieliliśmy na podtokeny: litera kolumny i cyfra rzędu rozdzielane były na dwie części, a każdy z tych podtokenów miał swój indeks w słowniku. W ten sposób ruch łączono w sekwencję długości zwykle 2 lub 4 tokenów.

Sekwencja wejściowa do transformera powstawała z połączenia tokenów opisujących bieżący stan gry (FEN). W naszym podejściu ograniczyliśmy się do stanu bieżącego (pełnego opisu FEN-a), traktując zadanie jako predykcję kolejnego ruchu wyłącznie na podstawie tego, co jest widoczne na planszy. Wyjście modelu ma postać jednego tokenu (lub sekwencji tokenów) odpowiadającego najlepszym ruchom zwróconym przez silnik; w końcowym etapie dekodowaliśmy go z powrotem do postaci tekstowej (notacja UCI lub odpowiednik dla warcab).

Architektura transformera Trasformer, który wykorzystaliśmy w projekcie, to wariant „decoder-only" inspirowany modelami GPT. Jego zadaniem było uczenie się mapowania sekwencji wejściowej (tokeny opisujące bieżący stan gry) na sekwencję wyjściową (token(y) ruchu). Całość została zaimplementowana jako klasa dziedziczącą po `pytorch_lightning.LightningModule`, co umożliwiło wygodne zarządzanie pętlą treningową, logowaniem oraz checkpointami.

Pierwszym etapem w module transformera jest warstwa embeddingu, która mapuje każdy token (liczony z zakresu od 0 do rozmiaru słownika minus jeden) na wektor o wymiarze $d_{\text{model}} = 512$. Następnie do każdego embeddingu dodawane jest pozycjonalne kodowanie (ang. *positional encoding*), dzięki któremu model wie, w której kolejności pojawiły się tokeny. Pozycjonalne kodowanie zostało zaimplementowane zgodnie z pomysłem z oryginalnej publikacji Vaswaniego i innych – obliczamy wektory sinusoida i cosinusoida dla kolejnych pozycji od 0 do maksymalnej długości sekwencji (w naszym przypadku ustalonej na 72 tokeny) i dodajemy je do embeddingu.

Rdzeń modelu składa się z sześciu kolejnych bloków transformera. Każdy blok zawiera maskowaną warstwę self-attention z ośmioma głowami oraz wielowarstwowy moduł feed-forward (w dwóch krokach liniowej transformacji z funkcją aktywacji ReLU pomiędzy). Po warstwie uwagi oraz po module feed-forward stosujemy normalizację warstwy (Layer Normalization) oraz dropout o prawdopodobieństwie odrzutu równym 0,1, aby zapobiegać nadmiernemu dopasowaniu. Maskowanie w warstwie self-attention jest konieczne po to, aby model podczas nauki predykcji następnego tokenu nie miał dostępu do elementów sekwencji wykraczających poza bieżącą pozycję (ang. *causal mask*). Dzięki temu sieć uczy się autoregresywnie generować ruch bazując tylko na tym, co jest już „widoczne".

Po przejściu przez wszystkie bloki transformera otrzymujemy tensor o kształcie $(\text{batch\_size}, \text{seq\_len}, d_{\text{model}})$. Aby wykonać predykcję ruchu, najpierw uśredniamy ten tensor po wymiarze sekwencji (tzw. pooling mean). W rezultacie otrzymujemy wektor o wymiarze $(\text{batch\_size}, d_{\text{model}})$, który następnie podajemy na w pełni połączoną warstwę liniową zwracającą logity dla każdego możliwego ruchu w słowniku (łącznie kilkanaście tysięcy tokenów ruchu). Do obliczenia funkcji straty wykorzystujemy entropię krzyżową (CrossEntropyLoss), porównującą przewidywany rozkład prawdopodobieństwa z rzeczywistym ruchem zapisanym w danych.

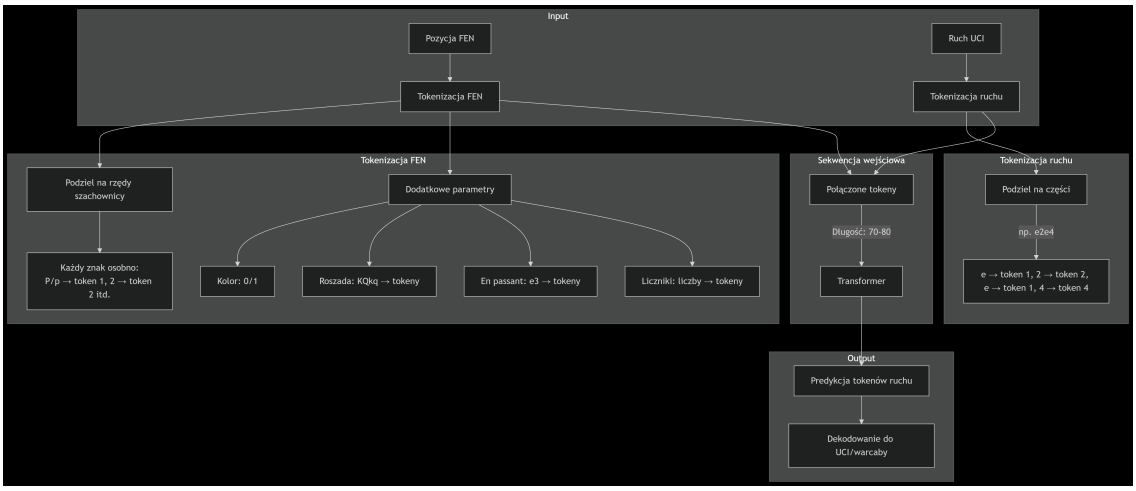

Figure 1: Schemat procesu tokenizacji danych wejściowych

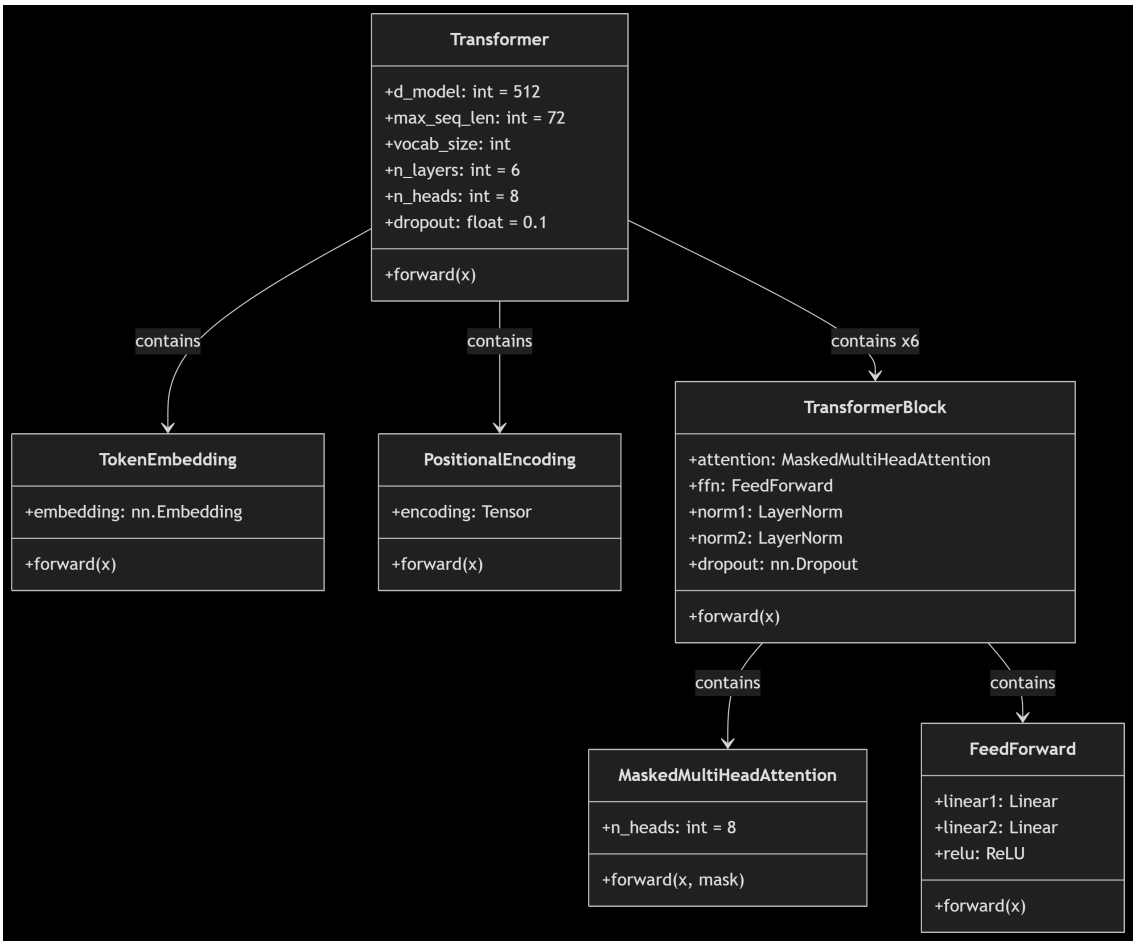

Figure 2: Schemat procesu tokenizacji danych wejściowych

2.3 Proces treningu

Trening modelu prowadziliśmy w środowisku PyTorch Lightning, co znacznie uprościło zarządzanie iteracjami treningowymi, walidacją oraz zapisem checkpointów. Dane wczytywane były z bazy SQLite, za każdym razem pobierany był kolejny chunk o wielkości 1000 plansz wraz z najlepszym ruchem uzyskanym poprzez zewnętrzny silnik szachowy bądź warcabowy. Przy każdym wywołaniu generatora pobieraliśmy partię (ang. batch) składającą się z 64 par $(\text{tokeny\_planszy}, \text{token\_ruchu})$. Na etapie tworzenia batcha każdą notację FEN zamienialiśmy na tensor liczb (tokenizacja opisana wcześniej), zaś ruch kodowaliśmy poprzez odwzorowanie notacji UCI na indeks w słowniku.

Jako optymalizator zastosowaliśmy AdamW z początkową wartością współczynnika uczenia $\text{lr} = 5 \times 10^{-5}$ oraz weight decay równym 1×10^{-2}. Plan treningu zakładał 20 epok, co w przybliżeniu przekładało się na 50 000 kroków (batchy), ponieważ całkowita liczba ocenionych pozycji wynosiła około pół miliona. Po każdej epoce obliczaliśmy stratę i metryki na zbiorze walidacyjnym. Jeśli przez dwie kolejne epoki nie obserwowaliśmy poprawy walidacyjnej straty, obniżaliśmy learning rate o czynnik 0,5. Dodatkowo zapisywaliśmy checkpointy modelu co dwie epoki oraz zawsze, gdy metryka Top-1 Accuracy na zbiorze walidacyjnym uległa poprawie.

Dzięki internemu logowaniu Lightninga mogliśmy w czasie rzeczywistym śledzić spadek straty i wzrost dokładności (Accuracy), a także liczbę nielegalnych ruchów zgłoszonych przez funkcję oceniającą. Każde przewidziane przesunięcie było weryfikowane względem aktualnych zasad gry – na przykład w szachach sprawdzaliśmy, czy ruch należy do zbioru legalnych w danej pozycji na podstawie obiektu `chess.Board` z biblioteki `python-chess`. W przypadku warcab rozpoznawaliśmy ruchy wieloskokowe i sprawdzaliśmy, czy nie dochodzi do naruszenia zasad bicie-ciąg.

2.4 Ewaluacja

Po zakończeniu treningu model zweryfikowaliśmy na wcześniej wydzielonym zbiorze testowym, stanowiącym około 10% wszystkich ocenionych pozycji i nigdy nieużytym w trakcie uczenia. Proces ewaluacji przebiegał dwuetapowo. Najpierw wczytywaliśmy najlepszy checkpoint (według metryki Top-1 Accuracy na walidacji). Dla każdej próbki obliczaliśmy prawdopodobieństwo wszystkich możliwych ruchów i porównywaliśmy wybrany przez model ruch z rzeczywistym ruchem silnika (Stockfish lub Scan Engine). Ponadto sprawdzaliśmy, czy sugerowany ruch jest legalny – w przypadku niezgodności liczyliśmy je jako *illegal moves* i logowaliśmy tę wartość.

Do oceny jakości modelu posłużyły następujące metryki:

- *Accuracy* – wartość miary oceny jakości klasyfikatora wieloklasowego.
- *Errors* – odsetek przypadków, w których przewidziany ruch nie odpowiadał wartości sugerowanej przez silnik
- *Cross-entropy loss* – średnia wartość funkcji straty na zbiorze testowym.
- *Illegal Moves Count* – liczba nielegalnych ruchów (predicted_move nie należący do zbioru legalnych), co sygnalizuje, na ile model nauczył się zasad gry, a nie tylko wzorców

z danych.

Wyniki ewaluacji zapisywaliśmy w pliku metrics.csv, dzięki któremu mogliśmy później wygenerować wykresy obrazujące spadek straty oraz wzrost dokładności w kolejnych epokach (przy użyciu biblioteki `matplotlib`), a także porównać osobno dokładność modelu dla szachów i dla warcab.

3 Rezultaty

Uczenie modelu okazało się bardzo czasochłonne. Przetworzenie 1000 planszy szachów, w dwóch epokach, zajmowało kilka minut. Dlatego, skonfigurowaliśmy nasze procesory graficzne tak, aby móc wykonywać na nich program Pythona. Po tym usprawnieniu, byliśmy w stanie, w krótszym czasie przetworzyć tyle samo plansz ale w 20 epokach. Mimo optymalizacji uczenie było powolne, accuracy stopniowo się poprawiało, jednak niewystarczająco szybko.

Otrzymane rezultaty okazały się dalekie od oczekiwań. Chociaż model nauczył się rozpoznawać podstawowe zależności na planszy i sporadycznie trafiał z dokładnym ruchem silnika, to ogólny poziom gry pozostawał niski. Szczególnie kłopotliwa okazała się wysoka stopa nielegalnych ruchów.

4 Wnioski

Główne przyczyny niezadowalających wyników można podsumować następująco:

1. **Brak czasu oraz sprzętu potrzebnego do wytrenowania modelu** Problem szachów oraz warcab bez przeszukiwania, jest bardzo złożony. Zdawaliśmy sobie z tego sprawę już na początku pracy nad projektem, jednak zabrakło nam czasu potrzebnego do treningu. Dodatkowo, nasze sprzęty nie są wystarczająco zaawansowane by wydajnie uczyć złożony model.

2. **Parametry nie są odpowiednio dopasowane do problemu. Występuje zbyt wysoki wymiar d_model.** Wybór $d=512$ przy stosunkowo niewielkim zbiorze spowodował, że model miał więcej parametrów niż dane były w stanie wiarygodnie wytrenować.

3. **Tablica logitów obejmująca *wszystkie* możliwe ruchy.** Przestrzeń wyjściowa licząca kilkanaście tysięcy tokenów powodowała rozrzedzenie prawdopodobieństwa i utrudniała uczenie. Lepszym rozwiązaniem byłoby wyjście modelu prezentujące planszę z wykonanym najlepszym ruchem.

Nauka wyciągnieta z projektu Pomimo napotkanych ograniczeń, realizacja projektu umożliwiła nam zdobycie istotnych kompetencji w zakresie praktycznego wykorzystania architektury transformerów. W szczególności, uzyskaliśmy doświadczenie w implementacji oraz trenowaniu modeli tego typu. Analiza rezultatów pozwoliła na lepsze zrozumienie znaczenia

doboru odpowiednich hiperparametrów, sposobu reprezentacji danych wejściowych i wyjściowych, a także wpływu ograniczeń sprzętowych na efektywność procesu uczenia.

Projekt ten może stanowić solidną bazę do dalszych, bardziej udanych prób stworzenia modelu zdolnego do gry w szachy, warcaby lub inne gry planszowe. Kod źródłowy, pipeline przetwarzania danych, oraz przyjęte rozwiązania architektoniczne mogą być punktem wyjścia do bardziej zoptymalizowanych i skutecznych podejść — zarówno przez nas, jak i przez innych badaczy czy studentów zajmujących się podobnymi zagadnieniami.

5 Bibliografia

- Główne źródło: Artykuł Grandmaster-Level Chess Without Search
  `https://arxiv.org/html/2402.04494v1`
- Inny szachowy Transformer:
  `https://github.com/sgrvinod/chess-transformers`
- Szachowy evaluator:
  `https://stockfishchess.org/`
- Dane szachowe:
  `https://www.kaggle.com/datasets/arevel/chess-games/data`
- Silnik warcabowy:
  `https://lidraughts.org/`