VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS KATEDRA

Dirbtiniai neuroniniai tinklai JavaScript kalbos kodui iš teksto generuoti

JavaScript programming language code generation with artificial neural networks from input text

Baigiamasis magistro darbas

Atliko:	Liudas Demikis	(parašas)
Darbo vadovas:	prof. dr. Olga Kurasova	(parašas)
Recenzentas:	dr. Linas Petkevičius	(parašas)

Vilnius – 2023

Santrauka

Darbe analizuojamos dirbtinių neuroninių tinklų galimybės generuoti JavaScript kalbos kodą. Skirtingos neuroninių tinklų architektūros geba generuoti paveikslėlius, žmonių portretus ar natūralią žmonių kalbą. Pasirinktą architektūrą reikės adaptuoti kodo generavimui, ištirti koks algoritmas tiksliausiai generuotų kodą atsižvelgdamas į gauto rezultato prasmę bei kalbos sintaksę. Galutinis darbo rezultatas bus įrankis su sąsaja, kurioje pagal įvesties tekstą naudotojas gaus neuroninio tinklo sugeneruotą JavaScript kalbos kodą.

Raktiniai žodžiai: dirbtiniai neuroniniai tinklai, dirbtinis intelektas, JavaScript kalba, mašininis mokymasis, kodo generavimas

Summary

Artificial neural networks ability to generate JavaScript programming language code will be analysed in this research work. Different neural network architectures are able to generate images, human portraits and natural language. Selected architecture will be adapted to code generation problem, different algorithms will be evaluated to get the most meaningful and syntactically correct results. Final result of this work will be a tool with an interface where one will be able to get blocks of code generated by neural network by his input.

Keywords: artificial neural networks, artificial intelligence, JavaScript programming language, machine learning, code generation

Turinys

Sąvokos ir apibrėžimai

Babel – transpiliavimo įrankis, skirtas *JavaScript* programavimo kalbai.

Transpiliavimas – procesas panašus į kompiliavimą, kai viena kodo sintaksė pakeičiame į kitą, senesnę, dažniausiai skirta senų naršyklių palaikymui įgyvendinti.

JavaScript – programavimo kalba skirta tiek kliento dalies, tiek serverio dalies programavimui.

Code editor – programavimo aplinka, kurioje rašomas kodas.

IDE – integruota programavimo aplinka, dažnai skirta vienai konkrečiai kalbai bei turinti svarbiausius kalbą palaikančius įrankius kaip kompiliatorius.

ReactJs – kliento dalies *JavaScript* kalbos biblioteka.

NodeJs – Ssrverio dalies *JavaScript* veikimo aplinka parašyta su *C++* kalba.

EcmaScript – *JavaScript* kalbos standartus apibrėžiantis dokumentas.

ExpressJs – *JavaScript* kalbos serverio dalies karkasas, skirtas kurti internetinius serverius.

ShadowDOM – mechanizmas kuriuo *JavaScript* po vartotojo veiksmų atnaujina ne visą internetinę svetainę, bet tik reikiamas vietas.

JSX – ReactJs bibliotekos sintaksė apjungianti *JavaScript* bei *HTML*.

HTTP API – internetinio protokolo *HTTP* aplikacijos programinė sąsaja.

HTML – sintaksė kuria aprašoma internetinių svetainių struktūra.

CSS – sintaksė kuria aprašomas internetinių svetainių stilius.

NLTK – *Python* biblioteka darbui su natūralia kalba.

Android – telefonų operacinė sistema.

IOS – *Apple* telefonų operacinė sistema.

Github repozitorija – kodo saugykla *Github* svetainėje.

Esprima – Įrankis generuojantis abstrakčius sintaksės medžius *JavaScript* kalbai

Python – programavimo kalba

Tensorflow – biblioteka naudojama dirbtinio intelekto modelių kūrimams ir tyrimams *Google gcloud* – debesų kompiuterijos sprendimas

Iliustracijų sąrašas

Lentelių sąrašas

Įvadas

JavaScript programavimo kalba yra būtina interaktyvių internetinių svetainių dalis. Dauguma sukurtų internetinių svetainių turi daug pasikartojančių panašumų ne tik kode, bet ir galutiniame rezultate. Straipsnyje [SKS19] netgi aprašomas pačių internetinių svetainių kūrimas pagal įvesties tekstą ar įkeltą dizaino maketą. Nuo *JavaScript* programavimo aplinkos *NodeJs* atsiradimo 2009 metais ši kalba naudojama ir serverio dalies kodui rašyti. *JavaScript* kalba yra standartizuota dokumento pavadinimu *EcmaScript*. *JavaScript* turi daug populiarių kliento dalies bei serverio dalies bibliotekų ir karkasų. Šie įrankiai padeda išspręsti pasikartojančias problemas, su kuriomis susiduria visi internetinių puslapių kūrėjai. Pavyzdžiui, *ReactJs* biblioteka leidžia programuotojams našiau valdyti kodo būsenas bei atmintį, taip pat dėl turimo funkcionalumo *ShadowDOM* padidina internetinės svetainės dinamiškumą. Karkasas *ExpressJs* serverio dalyje leidžia kurti internetines svetaines įgyvendinant vieną svarbiausių dizainų šablonų – tarpines funkcijas. Įvairių bibliotekų bei karkasų panaudojimas reikalauja nemažai šabloninio kodo norint paleisti juos kuriamoje programoje. Pavyzdžiui *ReactJs* programėlėje rašoma kiek kitokia sintaksė nei paprastas *JavaScript* kodas – *ReactJs* naudoja *JSX* sintaksę. Keletas dažnai pasikartojančių šabloninio kodo panaudojimo atvejų:

1. Transpiliavimas (angl. *transpiling*) – procesas, kai vieną kalbos versiją verčiame į senesnę. Tai reikalinga norint, kad kuriama internetinė svetainė kliento dalyje veiktų ant senesnių naršyklių, pavyzdžiui *Internet Explorer* 11 versijos. Šis transpiliavimo procesas dažniausiai yra šabloninis įrankio *Babel* panaudojimas, kopijuojamas iš vieno projekto į kitą su mažais pakeitimais.

2. Naujo komponento sukūrimas ir priregistravimas prie taikomosios programos naviguojamų puslapių. Norint pridėti naują puslapį į programą reikia keisti nemažai vietų kopijuojant jau egzistuojantį komponentą, kurio aprašyta logika gali visiškai nesutapti su norimu. Dažniausiai nukopijavus tokį kodą ištrinama daugiau eilučių nei paliekama.

3. Saugumo žetonų valdymas tiek kliento dalyje, tiek serverio dalyje. Neretai aplikacijos turi resursų nepasiekiamų neautorizuotiems vartotojams. Paslapčių (angl. *secrets*) saugojimo kode problema yra aprašoma straipsnyje [SSS20].

4. Naujos duomenų bazės lentelės ar kolekcijos sukūrimas, migracija. Programuotojai visada kopijuoja anksčiau parašytas schemas apibrėžiančias lenteles kode ir pakeičia įvairius laukus bei atributus siekdami pritaikyti naujai lentelei.

JavaScript programuotojų bendruomenė yra sukūrusi daug viešai prieinamų bibliotekų ir karkasų, kurie yra nuolatos atnaujinami ir tobulinami. Dalis egzistuojančių bibliotekų yra komandinės eilutės sąsajos pavidalo įrankiai skirti generuoti tuščius komponentus, puslapius, *HTTP API* kelius, tuščias *Babel* ar *Webpack* konfigūracijas. Tačiau tokie įrankiai nėra pritaikyti sudėtingesnėms konfigūracijoms, konkretiems panaudos atvejams ar veikiantys bendrai visam *JavaScript* – jie būna pririšti prie konkrečios technologijos.

Norint generuoti kalbos kodą susijusį su konkrečiais panaudos atvejais, neapsiriboti tik tuščiais komponentais ar šabloninėmis konfigūracijomis reikėtų ištirti *JavaScript* kalbos savybes bei pasinaudoti dirbtinio intelekto algoritmais. Šiame darbe bus tiriama kaip būtų galima generuoti *JavaScript* kalbos kodą pagal įvesties tekstą. Kadangi *JavaScript* programavimo kalba yra interpretuojama, o ne kompiliuojama, pasirinkti šie du kodo generavimo analizavimo keliai:

1. Skaidyti kodą į abstraktų sintaksės medį.
2. Generuoti kodą kaip natūralią kalbą skaidant jį į paprastus žodžius.

Kaip jau buvo minėta, atliekant tyrimą bus pasitelkiamas dirbtinis intelektas (angl. *artificial intelligence*), o tiksliau – mašininio mokymosi (angl. *machine learning*) šaka – dirbtiniai neuroniniai tinklai (angl. *artificial neural networks*). Šiame darbe bus apžvelgti trys neuroninių tinklų algoritmai, gebantys generuoti natūralią kalbą arba kodą remiantis abstrakčiais sintaksės medžiais:

1. Rekurentiniai neuroniniai tinklai ([HS97])
2. Generatyviniai besivaržantys neuroniniai tinklai ([GPM+14])
3. Transformerių neuroniniai tinklai ([VSP+17])

Darbo tikslas: Išanalizuoti dirbtinių neuroninių tinklų modelių savybes ir kaip jos daro įtaką rezultatų tikslumui sprendžiant *JavaScript* kalbos kodo iš teksto generavimo uždavinį, pasiūlyti šiam uždaviniui spręsti modelį ir sukurti įrankį.

Darbo uždaviniai:

1. Skirtingų dirbtinių neuroninių tinklų architektūrų analizavimas ir palyginimas.
2. Kodo generavimo metodų tyrimas (abstraktūs sintaksės medžiai, gramatikos žetonų žodynai).
3. Dirbtinio neuroninio tinklo architektūros adaptavimas pasirinktam kodo generavimo metodui.
4. Duomenų surinkimas iš atvirojo kodo platformų.
5. Duomenų apdorojimas, paruošimas dirbtinio neuroninio tinklo mokymui.
6. Dirbtinio neuroninio tinklo apmokymas.
7. Rezultatų testavimas bei testavimo automatizavimas.
8. Komandinės eilutės sąsajos sukūrimas.

1. Literatūros analizė

1.1. Abstraktūs sintaksės medžiai

Abstraktūs sintaksės medžiai (angl. *abstract syntax trees, ASM*) yra aukšto lygio kodo reprezentacija. Šis įrankis yra vadinamas medžiu, todėl, kad yra suformuojamas remiantis to paties pavadinimo duomenų struktūra – medžiu (angl. *tree*). Medis yra paremtas hierarchine struktūra, tai yra jei kode aprašyta funkcija turi kintamųjų, tie kintamieji priklausys funkcijos viršūnei. Straipsnyje [RM17] apibūdinami skirtingi lapų tipai - kiekviena medžio viršūnė gali būti primityvaus arba sudėtinio tipo. Straipsnyje taip pat pabrėžiama, kad primityvūs tipai tokie kaip *string, number* bus reprezentuojami primityviomis viršūnėmis, o sudėtines viršūnes sudarys funkcijos, kodo blokai ar klasės. Kiekviena sudėtinė viršūnė savyje gali turėti kitas sudėtines ar primityvias viršūnes, tačiau primityvi viršūnė savo kodo atšakų turėti nebegali.

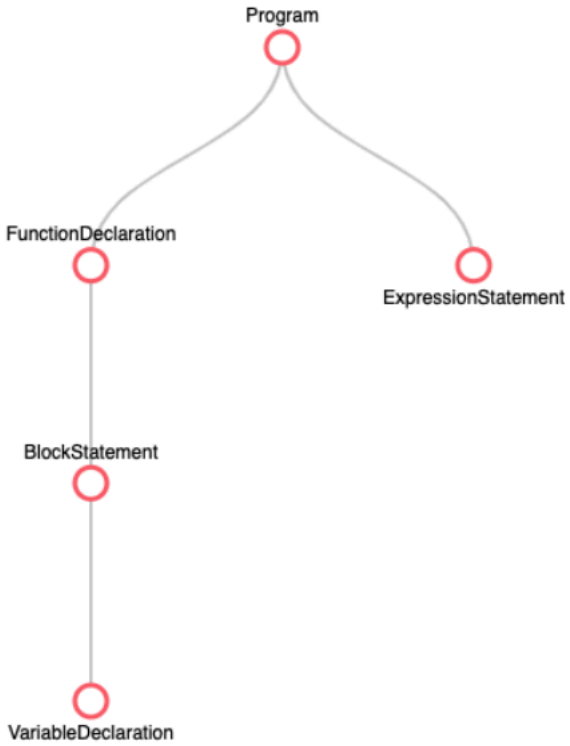
1.1.1. Abstraktaus sintaksės medžio pavyzdys

Pavyzdyje, pateiktame 1 pav., yra paprastas *JavaScript* kodas kuriame aprašyta viena funkcija bei joje esantis kintamasis. Iš šio kodo sugeneruotas abstraktus sintaksės medis pateiktas 2 pav. esančiame pavyzdyje.

```
1  function add() {
2    const a = 'example 1';
3  }
4
5  add();
6
```

1 pav. *JavaScript* funkcijos ir kintamojo deklaracijos pavyzdys

Abstrakčių sintaksės medžių sudarymas neturi vieningų taisyklių skirtingoms programavimo kalboms, tačiau yra tiriami universalūs ASM. Straipsnyje [LGC+20]; minima, kad *Java* kalbos išraiškos apibrėžiančios funkcijas, ciklus ir kiti kodo elementai priklausantys nelapinėms viršūnėms sugeneruoja viršūnes, turinčias daugiau informacijos apie pačią sintaksę. Tai galime matyti ir pirmame *JavaScript* ASM pavyzdyje (2 pav.) – funkcijos apibrėžimas sukuria bloko deklaracijos viršūnę, kurioje apibrėžiamas paprastas kintamasis yra lapinė viršūnė – negali ir neturi jokių lapų po savimi.



2 pav. JavaScript ASM pavyzdys

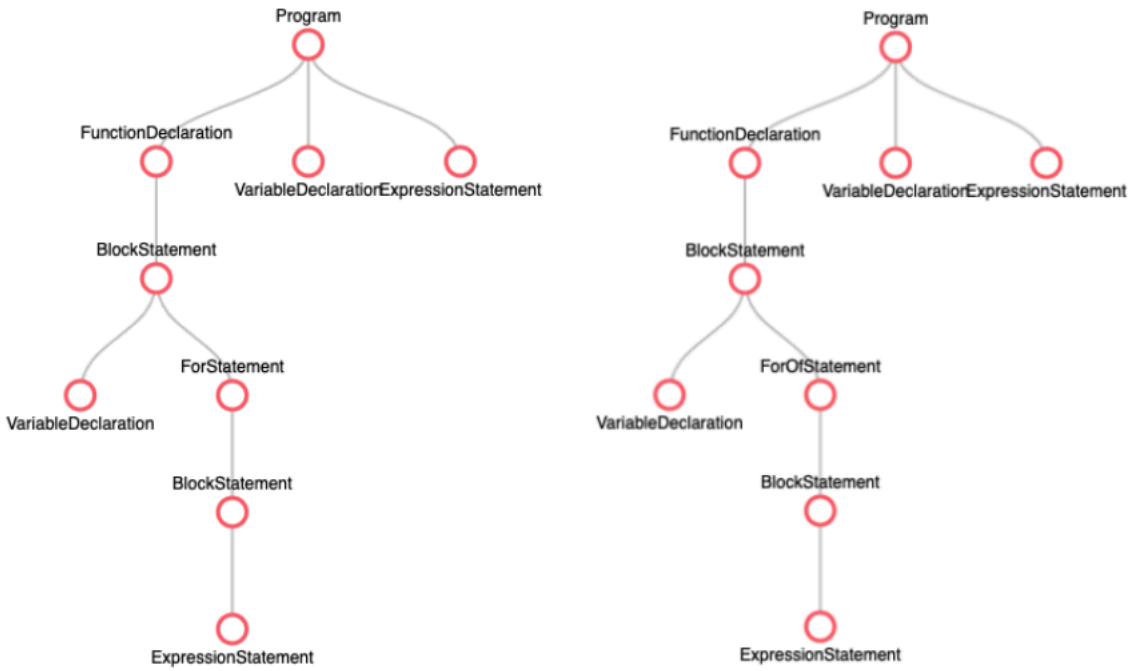
1.1.2. Abstraktaus sintaksės medžio pranašumas panašiuose kodo fragmentuose

Darbe [ABL+18] yra nurodomas ASM privalumas parodant du skirtingus *Java* kodo fragmentus, kurie sugeneruoja beveik identiškus medžius – skiriasi tik viena viršūnė. Galime įsitikinti, kad tai galioja ir *JavaScript* kalbai (3 pav. ir 4 pav.).

```
1   function arraySum(numbers) {
2     let sum = 0;
3
4     for (let i = 0; i < numbers.length; i++) {
5       sum += numbers[i];
6     }
7   }
8
9
10  const numbers = [3, 4, 5];
11
12  arraySum(numbers);
13
```

```
1   function arraySum(numbers) {
2     let sum = 0;
3
4     for (let i of numbers) {
5       sum += i;
6     }
7   }
8
9
10  const numbers = [3, 4, 5];
11
12  arraySum(numbers);
13
```

3 pav. *JavaScript* skirtingos sumos operacijų įgyvendinimo kodas



4 pav. *JavaScript* skirtingos sumos operacijų ASM

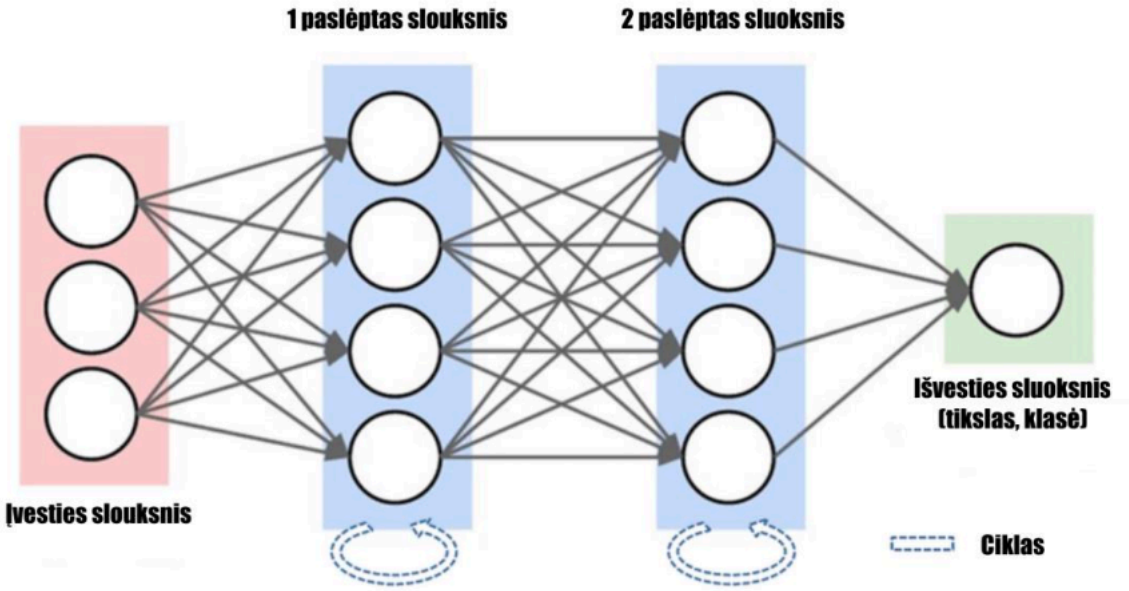
Straipsnyje [ABL+18] pabrėžiama, kad toks pavyzdys parodo ASM generavimo pranašumą prieš žetonais paremtą kodo analizę – mašininio mokymosi algoritmas sugebės atpažinti pasikartojančius kodo kelius naudodamasis ASM.

1.2. Rekurentiniai neuroniniai tinklai kalbos generavimui

1.2.1. Rekurentinių neuroninių tinklų architektūra

Rekurentiniai neuroniniai tinklai – tai neuroniniai tinklai, kurių architektūrą sudaro kryptinis grafas. Šių neuroninių tinklų svarbi savybė yra ta, kad jie gali turėti vidinę būseną – atmintį. Rekurentiniai neuroniniai tinklai naudojami rašytinio teksto, garso atpažinimui. Šių tinklų panaudojimas sudėtingų sekų generavimui analizuojamas straipsnyje [Gra14]. Jame autorius teigia, kad *LSTM* neuroninių tinklų atmintis leidžia įsiminti priklausomybes ilgesnėse sekose – tai yra svarbi savybė generuojant kodo sekas.

Rekurentinių neuroniniai tinklai yra sudaryti iš kelių skirtingų sluoksnių (5 pav.). Skirtingi rekurentinių neuroninių tinklų sluoksniai priešingai nei kituose tinkluose yra priklausomi vieni nuo kitų, tarp skirtingų sluoksnių gali egzistuoti ir ciklinės priklausomybės, leidžiančios tinklui turėti atmintį.



5 pav. Rekurentinių neuroninių tinklų sluoksniai[1]

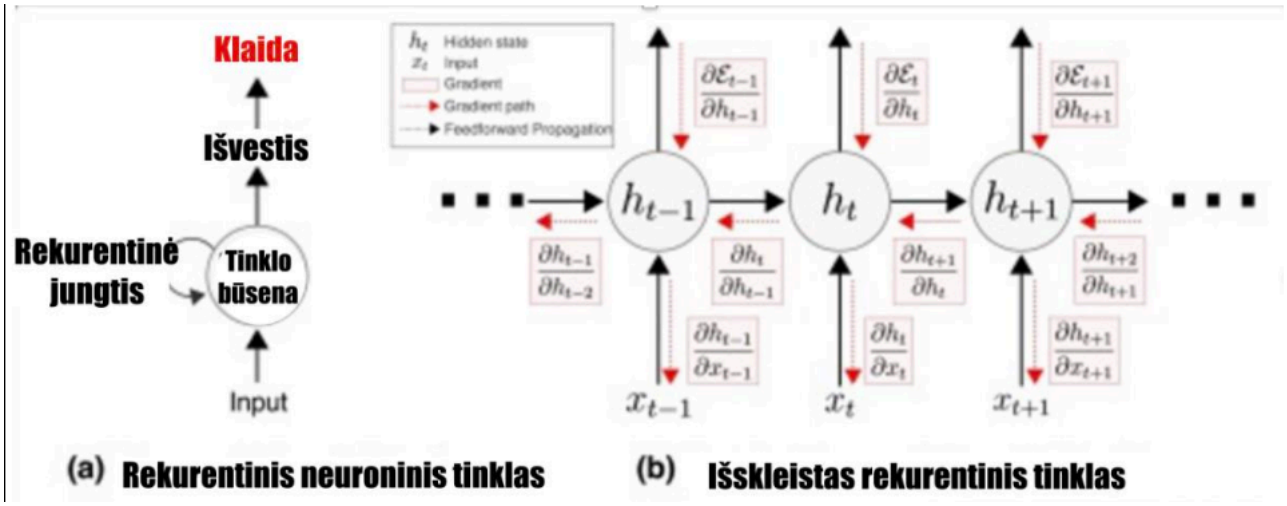
1.2.1.1. Atgalinės sklaidos algoritmas

Rekurentiniai neuroniniai tinklai naudoja atgalinės sklaidos per laiką (angl. *backpropagation through time, BPTT*) algoritmą, kuris šiek tiek skiriasi nuo įprasto atgalinės sklaidos algoritmo (angl. *backpropagation, BP*). Abiejų algoritmų principas yra toks pats – modelis save apmoko skaičiuodamas nuostolio vertę nuo išvesties iki įvesties sluoksnio. Moksliniame straipsnyje [She20] paaiškinama, kad nors atgalinės sklaidos algoritmas naudojamas tik tiesioginio sklidimo neuroniniuose tinkluose, kaip vieno sluoksnio perceptronas, šį algoritmą taip pat galima pritaikyti rekurentiniams neuroniniams tinklams. *BPTT* algoritmas bei rekurentinių tinklų išskleidimas (6 pav.) aprašomas straipsnyje [T P19]. Keturi pagrindiniai algoritmo žingsniai:

1. Neuroniniam tinklui pateikti įvesties ir išvesties porų sekas skirtingais laiko žingsniais;
2. Išskleisti (6 pav. b) neuroninį tinklą, paskaičiuoti bei susumuoti klaidas kiekviename laiko žingsnyje;
3. Atgal suskleisti (6 pav. a) neuroninį tinklą ir atnaujinti svorius siekiant sumažinti klaidas;
4. Kartoti algoritmą.

Pati išskleidimo operacija yra atliekama K serijai veiksmų, o kiekvienas laiko žingsnis žymimas n. Rekurentinio neuroninio tinklo išskleidimas yra tiesiog ciklinės operacijos panaikinimas perrašant neuroną K kartų. 6 pav. a dalyje parodytas neuroninis tinklas turintis ciklinę priklausomybę, o 6 pav. b dalyje galima matyti tą patį tinklą skirtingais laiko žingsniais ir vieno neurono priklausomybę nuo kito.

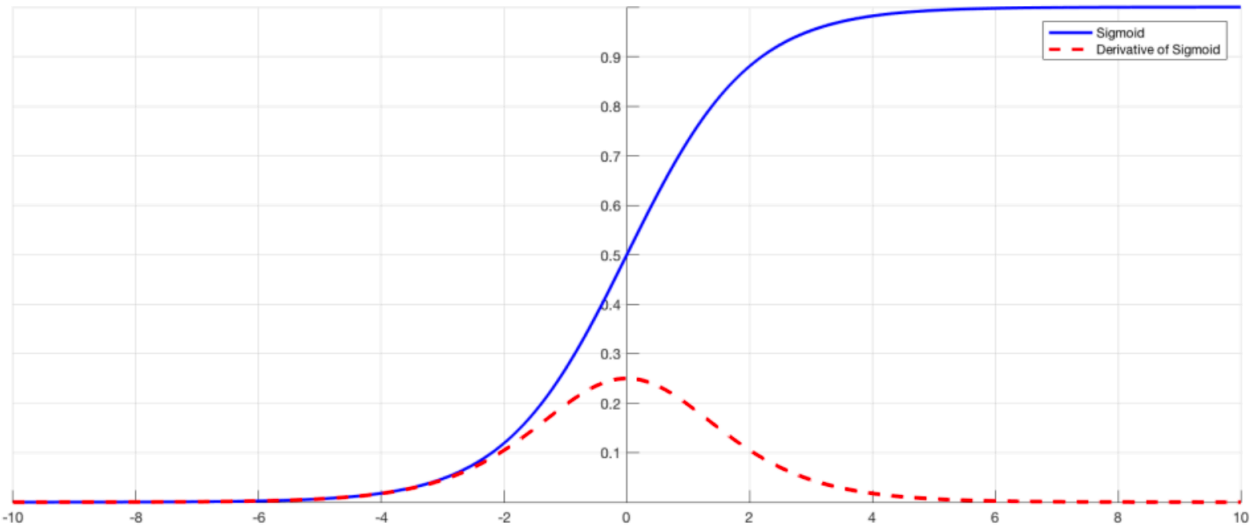
[1] `https://deepai.org/machine-learning-glossary-and-terms/recurrent-neural-network`



6 pav. Rekurentinis tinklas (a) ir išskleistas rekurentinis tinklas (b)

1.2.1.2. Gradientinis nusileidimas

Atgalinės sklaidos algoritmai naudojasi gradientinio nusileidimo metodu (angl. *gradient descent*). Gradientinis nusileidimas nusako greičiausią funkcijos augimo kryptį. Siekiant sumažinti neuroninio tinklo klaidos dydį naudojamas antigradientas – jis skirtas surasti funkcijos žemiausią tašką. Tačiau šio metodo naudojamas sukelia problemą – nykstančio gradiento. Kadangi gradientinio nusileidimo metodas naudoja funkcijos išvestinę, naudojantis tam tikromis aktyvacijos funkcijomis, pavyzdžiui sigmoidine, gradientas gali artėti prie nulio ir galų gale jį pasiekti. Tai gali visiškai sustabdyti neuroninio tinklo mokymąsi (7 pav.).



7 pav. Sigmoidinės funkcijos reikšmė ir išvestinė[2]

Paprasčiausias šios problemos sprendimas yra kitos aktyvacijos funkcijos panaudojimas, kuri nesukuria mažų išvestinių reikšmių, pavyzdžiui – *ReLU*. Tačiau toliau bus nagrinėjama kita rekurentinių neuroninių tinklų architektūra, kuri išsprendžia nykstančio gradiento problemą. Tai ilgos

[2] `https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e`

trumpalaikės atminties modelis (angl. *Long short-term Memory – LSTM*) pirmą kartą pristatytas 1997 metais straipsnyje [HS97].

1.3. LSTM architektūra

1.3.1. Sklendės

LSTM neuroniniai tinklai sudaryti iš trijų pagrindinių blokų arba sklendžių (vartų):

1. Įvesties – nusprendžia ar celės reikšmė turėtų būti atnaujinta;
2. Išvesties – nusprendžia ar celės reikšmė turėtų būti išvesta/matoma kitiems neuronams;
3. Pamiršimo – nusprendžia ar celės reikšmė turėtų būti pamiršta, tai yra – nustatyta į 0 reikšmę.

Kaip celės reikšmės yra atnaujinamos aprašoma straipsnyje [She20]. Jame pateikiamame pavyzdyje konkreti celė yra išskleidžiama *K* kartų. Celė *t* žingsnyje iš *K* laiko žingsnių priims t-ąjį įvesties vektoriaus $\vec{x}$ parametrą x_t. Vidinė celės būsena yra paskaičiuota pagal įvesties vektorių saugoma kitame būsenų (angl. states) vektoriuje $\vec{s}$. Šį vektorių pasiekti gali tik pati celė ir kiti tolimesni žingsniai (n + 1) toje pačioje celėje gali pasiekti praėjusias būsenų vektoriaus reikšmes. Išorinėms reikšmėms, kurios pasiekiamos kitoms celėms yra atskiras vektorius – $\vec{v}$.

Straipsnyje [Tha18] aprašytos kiekvienos iš blokų turi lygtį, pagal kurią apskaičiuojamos celės sklendės reikšmės:

1. Įvesties sklendės lygtis: $i_t = \sigma(w_i[h_{t-1}, x_t] + b_i)$
2. Pamiršimo sklendės lygtis: $f_t = \sigma(w_f[h_{t-1}, x_t] + b_f)$
3. Išvesties sklendės lygtis: $o_t = \sigma(w_o[h_{t-1}, x_t] + b_o)$

Simbolių reikšmės:

- i_t - įvesties sklendės reikšmė
- f_t - pamiršimo sklendės reikšmė
- o_t - išvesties sklendės reikšmė
- σ - *sigmoidinė* aktyvacijos funkcija
- w_x - x sklendės neurono svoris
- h_{t-1} - praėjusio laiko žingsnio to paties neurono išvesties reikšmė
- x_t - dabartinio laiko žingsnio įvesties reikšmė
- b_x - poslinkis
- *t* - laiko žingsnis

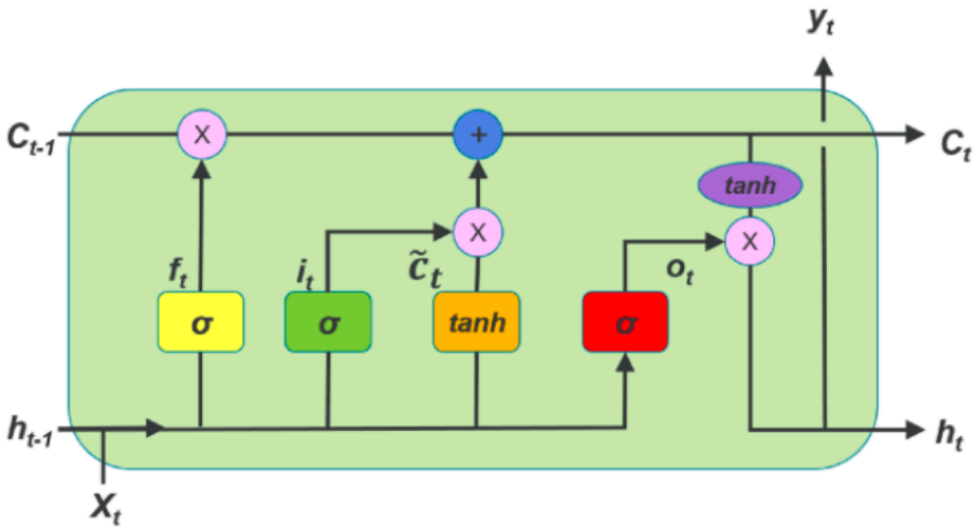
Taip pat, atskiros lygtys yra celės būsenai, kandidatei celės būsenai bei galutinei išvesties reikšmei:

1. Celės būsena: $\tilde{c_t} = \tanh\left(w_c[h_{t-1}, x_t] + b_c\right)$;
2. Celės būsenos kandidatė: $c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c_t}$;
3. Išvesties sklendės lygtis: $h_t = o_t \cdot \tanh(c^t)$.

Simbolių reikšmės:

- $\tilde{c_t}$ – celės būsena šiuo laiko žingsnyje t;
- c_t – celės būsenos kandidatė laiko žingsnyje t;
- tanh – hiperbolinio tangento funkcija.

Sąryšį tarp šių lygčių galima matyti 8 pav., kur pavaizduotas *LSTM* vienas blokas vieno laiko žingsnyje *t*.



8 pav. *LSTM* blokas *t* laiko žingsnyje ([Tha18])

1.3.1.1. Aktyvacijos funkcijos

Straipsnyje [She20] paaiškinama, kokios *tanh* funkcijos savybės pagrindžia jos panaudos atvejį skaičiuojant celės būseną bei išvesties reikšmę. Hiperbolinio tangento funkcijos turi tiek neigiamas, tiek teigiamas ribas. Tai padeda apriboti ir celės būsenos bei išvesties rezultatus. Sigmoidinė aktyvacijos funkcija naudojama, nes jos rezultatas visada yra intervale tarp 0 ir 1. Tai leidžia valdyti informaciją išeinančią per skirtingas sklendes.

LSTM blokai savyje išlaiko trumpalaikę būseną taip išspręsdami rekurentinių neuroninių tinklų problemą ilgose sekose per ilgai išlaikyti priklausomybes, kurios galimai nebėra aktualios. Taip pat, kaip buvo minėta anksčiau, *LSTM* išsprendžia nykstančio gradiento problemą pamiršimo sklendės pagalba – jų reikšmė apskaičiuojama naudojantis sigmoidine funkcija kuri naudojant kartu su *tanh* funkcija, leidžia išvengti nykstančio gradiento problemos.

1.3.2. ASM generavimas su LSTM

Mokslininkai Yin and Nubig dviejuose straipsniuose [YN17] bei [YN18] aprašo *LSTM* neuroninių tinklų galimybes generuoti programinį kodą generuojant abstraktų sintaksės medį.

1.3.2.1. ASM generavimas naudojimasis veiksmų kūrimu

Pirmajame darbe, [YN17], autoriai siūlo generuoti kodą remiantis jų sukurtu gramatikos modeliu, skirtu generuoti ASM iš tam tikrų veiksmų (angl. *actions*) arba žetonų (angl. *tokens*). Darbe toliau detaliau aprašomi kas yra veiksmai ir žetonai bei kaip tai sukuria medį. Antrame skyriuje minėtos dvejopos medžių viršūnės – primityvaus ir sudėtinio tipo – glaudžiai susijusios su mokslininkų pasirinktomis metodikomis – pritaikyti taisyklę veiksmas (angl. *Apply Rule action, ApplyRule*) bei žetono generavimo veiksmas (angl. *Generate Token Action, GenToken*). *ApplyRule* – tai veiksmas kuriantis medžio viršūnes iš viršaus į apačią ir iš kairės į dešinę. Kai neuroninis tinklas sukuria viršūnes, toliau pereinama prie *GenToken* veiksmo. Šis veiksmas užpildo viršūnes prieš tai apibrėžtomis konstantomis – pavyzdžiui, kai *ApplyRule* sukuria viršūnę kurioje yra apibrėžiamas kintamasis – *GenToken* veiksmas užpildys kintamojo viršūnę žetonais. Viena viršūnė gali turėti vieną arba kelis žetonus, taip pat mokslininkai naudoja specialų žetoną *</n>* pažymėti viršūnės pabaigą. Šie žetonai, pavyzdžiui funkcijų pavadinimai ar kintamųjų reikšmės, gali būti išgaunami dvejopai. Arba paimami iš prieš tai apibrėžto žodyno, arba generuojami iš užklausos įvesties natūralios kalbos.

Realizuoti šį funkcionalumą mokslininkai naudoja užkodavimo-dekodavimo (angl. *Encoder-Decoder*) neuroninį tinklą. Šios neuroninio tinklo architektūros principas pavaizduotas 9 pav. *Encoder* neuroniniam tinklui naudojama *Bidirectional LSTM* architektūra, o *Decoder* naudojama paprasta *LSTM* architektūra.

9 pav. *Encoder-Decoder* veikimo principas[3]

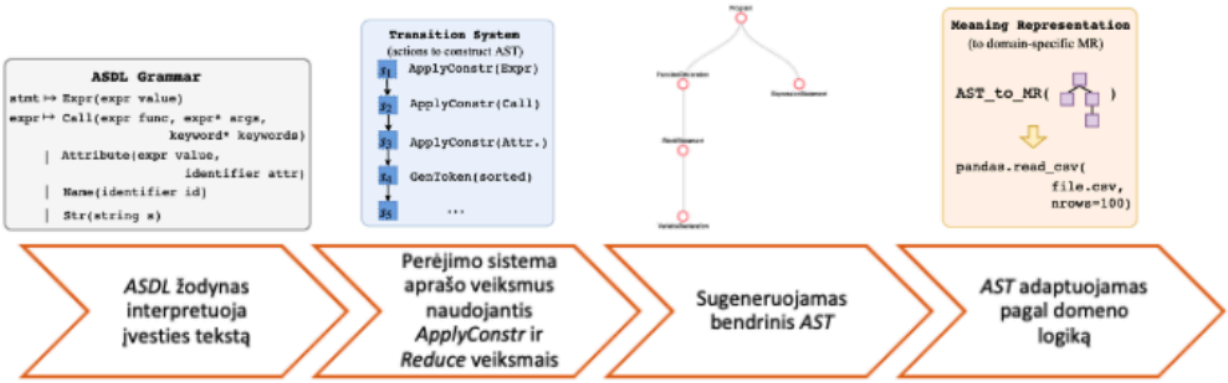
1.3.2.2. Tranx architektūra

Metais vėliau išleistame straipsnyje [YN18] mokslininkai aprašo naują jų sukurtą būdą generuoti kodą. Jį pavadino *Tranx* – perėjimais paremtas abstrakčios sintaksės transliavimas (angl. *Transition-based abstract syntax parser*) (10 pav.). Šiame metode yra paimamas natūralios kalbos programuotojo įvestas tekstas bei naudojantis jų sukurta perėjimo sistema aprašomi veiksmai skirti sugeneruoti abstraktų sintaksės medį. Kaip straipsnio [YN18] autoriai sako, patys įkvėpti savo pirmojo darbo naudojasi perėjimo sistema generuoti eilę medžio kūrimo veiksmų. Autoriai taip pat naudoja abstraktų sintaksės aprašymo kalbos žodyną (angl. *abstract syntax description language*,

[3] `https://medium.com/nerd-for-tech/encoder-decoder-model-for-machine-translation-8a90be12ac32`

ASDL) – skirtą generuoti abstrakčius sintaksės medžius. Šiame tinkle jie atsisako *ApplyRule* veiksmo ir vietoj to pristato du naujus: pritaikyti konstruktorių (angl. *Apply Constructor, ApplyConstr*) – parenka ir pritaiko konstruktorių iš *ASDL* ir sumažinimo (angl. *Reduce*) – pažymi, kad konkrečios medžio šakos viršūnių generavimas turėtų būti baigtas. Paskutiniame žingsnyje, jau sugeneravus ASM vykdomas reikšmės reprezentavimo žingsnis (angl. *Meaning Representation*, *MR*), jis leidžia adaptuoti ASM pagal įvesties tekstui artimą domeno logiką.

Mokslininkai nepakeitė neuroninio tinklo architektūros ir naudojo tą pačią, kaip ir straipsnyje [YN17], *Encoder-Decoder* 10 pav. struktūrą, kur *Encoder* tinklas yra dvikryptis (angl. *Bidirectional*) *LSTM*, o *Decoder* paprastas *LSTM*.



10 pav. *Tranx* sistemos veikimo žingsniai ([YN18])

1.3.2.3. Abstraktų sintaksės medį generuojančių modelių rezultatai

Straipsnio [YN18] autoriai pabrėžia, kad jų sukurta sistema geba generuoti skirtingų programavimo kalbų kodą – jų pagrindinė kalba yra *Python*, tačiau neuroninis tinklas sugeba generuoti net ir *SQL* šeimos kalbų kodą. Abiejuose darbuose palyginami skirtingi kodo generavimo algoritmai. Mokslininkų skaičiavimu pirmasis neuroninis tinklas [YN17] geba generuoti kodą 71,6 % (11 pav. a) tikslumu (*Django Python* kodo karkasas). Antrame darbe [YN18] – *Tranx* algoritmas pasiekia 73,7 % (11 pav. b) to paties karkaso tikslumą, o tiksliausiai geba generuoti *TypeSQL* kodą – 82,6 %.

	HS		DJANGO	
	ACC	BLEU	ACC	BLEU
Retrieval System†	0.0	62.5	14.7	18.6
Phrasal Statistical MT†	0.0	34.1	31.5	47.6
Hierarchical Statistical MT†	0.0	43.2	9.5	35.9
NMT	1.5	60.4	45.1	63.4
SEQ2TREE	1.5	53.4	28.9	44.6
SEQ2TREE–UNK	13.6	62.8	39.4	58.2
LPN†	4.5	65.6	62.3	77.6
Our system	16.2	**75.8**	**71.6**	**84.5**

A

Methods	ACC.
Phrasal Statistical MT (Ling et al., 2016)	31.5
SEQ2TREE (Dong and Lapata, 2016)	39.4
NMT (Neubig, 2015)	45.1
LPN (Ling et al., 2016)	62.3
YN17 (Yin and Neubig, 2017)	71.6
TRANX (w/o parent feeding)	72.7
TRANX (w parent feeding)	**73.7**

B

11 pav. Straipsnių [YN17; YN18] rezultatų ištraukos

1.3.3. Kodo generavimas naudojantis žetonais su LSTM

Straipsnyje [SKS19] aprašytas automatinis internetinių svetainių kūrimo įrankis (12 pav.), kurio kodas yra sudarytas iš trijų dalių – *HTML*, *CSS* bei *JavaScript*. Kaip ir prieš tai nagrinėtuose *LSTM* straipsniuose autoriai naudoja *Encoder-Decoder* neuroninių tinklų modelį. Šiame darbe yra taikomi du skirtingi įvesties būdai:

1. Naudotojo įvedamas tekstas apibūdinantis norimą rezultatą;
2. Naudotojo įkeliamas paveiksliukas, kuris parodo kaip turėtų atrodyti rezultatas.

Abu metodai naudoja savybių požymių išskyrimas (angl. *feature extraction*), kurie ištraukia vartotojo norimas savybes bei charakteristikas. Paveiksliukų savybių ištraukimui straipsnio autoriai naudoja konvoliucinį neuroninį tinklą (angl. *convolutional neural network, CNN*), o įvesties teksto – *Python NLTK* biblioteką, kurios pagalba įvesties tekstas paverčiamas žetonais.

Įvesties tekstas paverstas žetonais yra pakeičiamas sinonimais, pasinaudojant specialia leksikos duomenų baze *NLTK* bibliotekoje. Po šio žingsnio naudojamas *LSTM* neuroninis tinklas skirtas atpažinti bendrą temą tarp turimų žetonų ir pasiūlyti vartotojui kelias internetinio puslapio temas paveiksliukų pavidalu.

Išskyrus požymius iš paveiksliuko kitas žingsnis yra pasinaudoti domenui būdingos kalbos žetonais (angl. *domain specific language, DSL*). Šiame žingsnyje yra gaunamas toks pats rezultatas, kaip ir naudojantis įvesties tekstu – žetonų rinkinys. Šie žetonai toliau naudojami *Decoder LSTM* tinkle.

12 pav. Automatinio svetainių kūrimo įrankių žingsniai

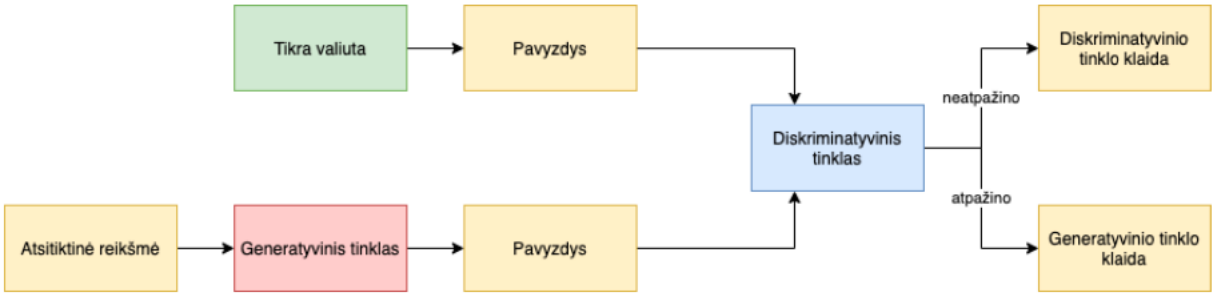
Straipsnio [SKS19] autoriai nepateikia tikslių rezultatų, bet teigia, kad sistema gali kurti internetines svetaines sugeneruodama veikiantį kodą. Taip pat teigia, kad sistemą galima plėsti iki

Android ar *IOS* programėlių kodo generavimo. Autoriai pabrėžia, kad naudojosi nedideliu duomenų rinkiniu ir sako, kad naudojantis daugiau duomenų apmokymui būtų galima pasiekti geresnius rezultatus.

1.4. Generatyvinių besivaržančių neuroninių tinklų savybės generuoti tekstą ir kodą

1.4.1. Generatyvinių besivaržančių neuroninių tinklų architektūra

Generatyviniai besivaržantys neuroniniai tinklai (angl. *Generative adversarial network, GAN*) – 2014 metais sukurtas karkasas, sudarytas iš dviejų neuroninių tinklų, kurie varžosi tarpusavyje varžybose, kur vieno iš tinklų pergalė yra kito pralaimėjimas. Šie neuroniniai tinklai pristatyti straipsnyje [GPM+14] grupės mokslininkų. Autoriai teigia, kad gilaus mokymosi (angl. *deep learning*) sėkmingiausios architektūros buvo tos, kurios naudojosi atgalinės sklaidos ir išmetimo (angl. *dropout*) algoritmais. Tačiau architektūros skirtos ką nors kurti, generuoti (angl. *generative*) nebuvo tokios sėkmingos dėl dviejų priežasčių: sudėtingumo įvertinti didelį kiekį skaičiavimų ir sudėtingumo pasverti tiesinių algoritmų panaudojimo naudą šių problemų sprendime.



13 pav. Generatyvinių besivaržančių neuroninių tinklų modelio pavyzdys

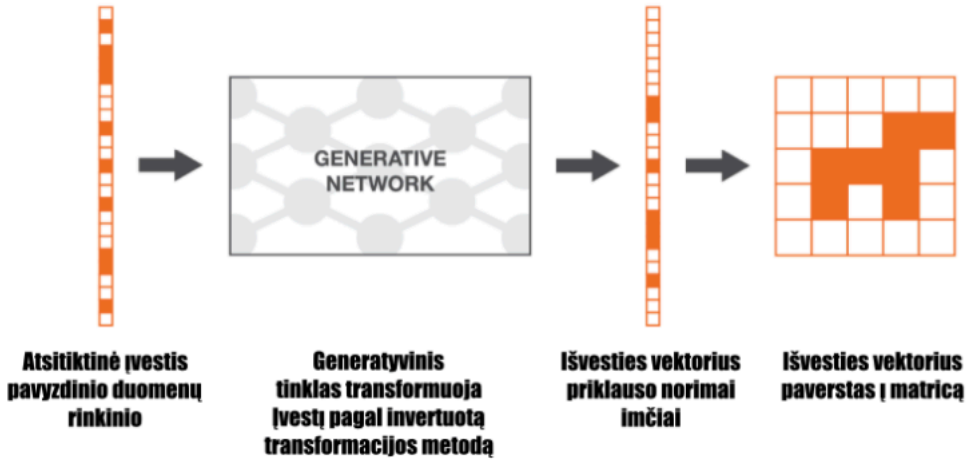
Straipsnio [GPM+14] siūlomas *GAN* modelis apibrėžia kovą tarp generuojančio bei jam besipriešinančio arba diskriminatyvaus (angl. *discriminator*) tinklo (13 pav.). Autoriai palyginimui siūlo generatyvinį modelį įsivaizduoti kaip vagis, kurių tikslas yra padirbti valiutą, o besipriešinantį modelį, kaip policiją, kuri bando atpažinti padirbtą valiutą. Šiame karkase vykstančios varžybos skatina abu modelius tobulinti savo metodus, kol generatyvinių tinklų rezultato diskriminatyvus tinklas nebesugeba atskirti nuo tikros valiutos. Straipsnyje [GPM+14] aprašomas karkasas naudoja kelių sluoksnių perceptroną (angl. *multilayer perceptron*) tiek generatyviniam tinklui, tiek diskriminatyviam tinklui.

1.4.1.1. Pseudo atsitiktinių reikšmių generavimas GAN įvesties reikšmėms

Straipsnyje [Roc18] aprašomi generatyvinių tinklų įvesties reikšmes ir paaiškina kaip veikia rezultatų generavimas iš atsitiktinių reikšmių. Pirmiausia straipsnio autorius pabrėžia, kad kompiuteriai geba generuoti pseudo atsitiktines (angl. *pseudo-random*) reikšmes, todėl generatyviniam besivaržančiam tinklui galima paduoti sugeneruotą atsitiktinę seką, kuri patenka į intervalą tarp

0 ir 1. Autorius aprašo du pažengusius pseudo atsitiktinių reikšmių generavimo algoritmus naudojamus sugeneruoti sekas generatyviniams tinklams:

1. Pavyzdžių atmetimo (angl. *rejection sampling*) – procesas, kuriam pateikiama duomenų aibė, iš kurios algoritmais atsitiktinai išmeta pasirinktinas reikšmes;
2. *Metropolis-Hasting* – algoritmas, kuris apskaičiuoja Markovo grandinę, kuri atitinka mūsų norimą duomenų rinkinį;
3. Atvirkštinės transformacijos metodas (angl. *inverse transform method*) – algoritmas, kuris paima atsitiktinį skaičių iš norimos sekos ir pritaiko transformacijos funkciją.



14 pav. Generatyvinio tinklo atsitiktinės skaičių generavimo funkcijų panaudojimo pavyzdys ([Roc18])

Toliau straipsnyje [Roc18] panaudojamas šunų paveikslėlių generavimo pavyzdys, kur generatyvinis tinklas naudojasi invertuotu transformacijos metodu ir geba iš atsitiktinių reikšmių generuoti norimus rezultatus (14 pav.).

1.4.1.2. GAN privalumai ir trūkumai

Generatyvinių besivaržančių neuroninių tinklų karkasas turi keletą privalumų ir trūkumų. Straipsnio [Roc18] autoriai kaip didžiausią privalumą generatyviniuose tinkluose išskiria atgalinės sklaidos panaudojimą gradiento skaičiavimui vietoj Markovo grandinių (šis procesas remiasi principu, jog praeitis yra nereikšminga numatant ateitį, taip darant neigiamą įtaką generatyvinio tinklo rezultatams). Kitas privalumas yra tai, kad generatyviniai tinklai atnaujinami diskriminatoriaus gradiento reikšme, o ne tiesioginiais įvesties duomenimis. Kaip minusą autoriai išskiria, kad generatyvinis tinklas negali per greitai pasiekti gerų rezultatų, turi nuosekliai būti atnaujinamas kartu su diskriminatyviu tinklu. Jei leidžiama generatyviniam tinklui išsiveržti į priekį, gali nukentėti rezultatų kokybė ateityje.

Generatyviniai besivaržantys neuroniniai tinklai dažniausiai naudojami paveiksliukų generavimui. Šiam darbui atlikti generatyvinis tinklas naudoja *CNN* architektūrą, o diskriminatyvusis

dažniausiai sprendžia klasifikavimo problemą, todėl naudoja daugiasluoksnio perceptrono (angl. *multilayer-perceptron*) architektūrą. Tačiau *GAN* yra pritaikomi ir teksto generavimui – natūralios kalbos generavimo problemai spręsti. *GAN* karkase generatyviniam tinklo algoritmui pasirinktus *LSTM* galima būtų generuoti ne tik natūralią kalbą, bet ir programinių kalbų kodą pasitelkiant prieš tai nagrinėtas architektūras.

1.4.2. GAN savybės generuoti tekstą

1.4.2.1. MaskGAN

Straipsnyje [FGD18] aprašomas generatyvinių besivaržančių neuroninių tinklų teksto generavimas naudojant mokymosi su paskatinimu algoritmą (angl. *Reinforcement Learning, RL*) generatyviniam tinklui. Autoriai argumentuoja šio algoritmo naudojimą, kad generuojant tekstą dėl jo diskrečios prigimties neįmanoma perduoti gradiento reikšmės iš diskriminatoriaus atgal į generatyvinį tinklą kaip įprasta *GAN* apmokyme. Būtent šiai problemai išspręsti autoriai siūlo naudoti mokymąsi su paskatinimu apmokant generatyvinį tinklą, bet diskriminatorių įprastai apmokant su stochastiniu gradientu.

Straipsnio [FGD18] autoriai iš karto pristato sunkumus su kuriais susiduria *GAN* generuodami tekstą. Generuojant paveikslėlius *GAN* gali skirtingose iteracijose sugeneruoti to paties objekto paveiksliukus – straipsnyje pateikiamas ugnikalnio pavyzdys – skirtingi to paties ugnikalnio kampai ar dydžiai. Diskriminatyvusis tinklas tokiu atveju klasifikuos šiuos paveiksliukus kaip realius nepaisant to, kad jie bus labai panašūs vieni į kitus. Tačiau generuojant tekstą toje pačioje situacijoje gausime labai panašius sakinius, kurie galbūt nebeturės prasmės. Straipsnyje siūlomas unikalus būdas išspręsti šią problemą – neuroninis tinklas apmokomas ne tiesiog generuoti tekstą, bet užpildyti tekste esančias tuščias vietas arba pataisyti neatitikimus.

Straipsnyje [FGD18] aprašomas įrankis pavadinimu *MaskGAN* naudoja informaciją priskiriant reikiamus žetonus tiek iš praeities (jau parašytos sakinio dalies), tiek ateities (numatomos likusios sakinio dalies). Tam įgyvendinti pasirinkta *seq2seq* (angl. *sequence to sequence*) architektūra. Kaip ir prieš tai apžvelgtuose *LSTM* modeliuose, ši architektūra yra *encoder-decoder* tipo. Čia *encoder* dalis atsakinga už užmaskuotos (sekos su redaguotomis arba ištrintomis reikšmėmis) sekos nuskaitymą ir šios informacijos perdavimą ateityje į *decoder* tinklą. *Decoder* dalis užpildo trūkstamas reikšmes. Autoriai pasirinko tokią pačią architektūrą ir diskriminatyviajam tinklui, kuris kaip įvestį gauną ir originalų kontekstą, prieš užmaskuojant pasirinktas reikšmes. Neturėdamas originalaus konteksto diskriminatorius gali klaidingai užskaityti nelogiškas sekas.

Toliau straipsnyje minima, kad ši architektūra nesėkmingai veikia su ilgomis sekomis arba dideliais žodynais ir adresuoja šią problemą. Ją autoriams išspręsti pavyko apmokymo fazėje – pirmiausia algoritmui pateikiama seka kuri gali turėti daugiausiai T kiekį žodžių. Kai neuroninis tinklas apmokomas pakankamai, kad peržengtų nurodytą kriterijų, T yra padidinamas vienetu. Taip neuroninis tinklas prieš generuodamas ilgas sekas išmoks dirbti su trumpomis.

Tinklas	Gramatika %	Tema %	Bendrai %
LM	15.3	19.7	15.7
MaskGAN	59.7	58.3	58.0
LM	20.0	28.3	21.7
MaskMLE	42.7	43.7	40.3
MaskGAN	49.7	43.7	44.3
MaskMLE	18.7	20.3	18.3
Real samples	78.3	72.0	73.3
LM	6.7	7.0	6.3
Real samples	65.7	59.3	62.3
MaskGAN	18.0	20.0	16.7

15 pav. *MaskGAN* rezultatai pagal žmonių vertinimus ([FGD18])

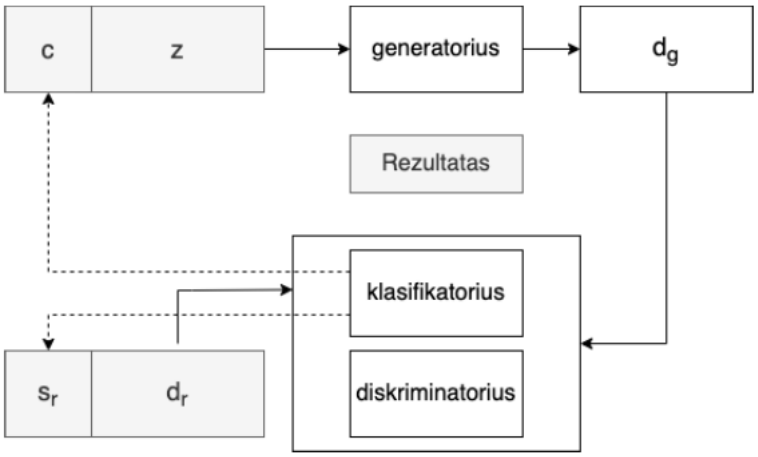
Straipsnio autoriai pasinaudodami tarptautinės duomenų bazės (angl. *International Movies Database IMDb*) tekstais palygino savo kuriamą *MaskGAN*, anksčiau sukurtą tinklą pavadinimu *LM* (angl. *Language Model*) ir trečią tinklą *MaskMLE*. 15 pav. pateikiami įvertinimai yra žmonių nežinančių tekstų kilmės vertinimai. Autoriai pasirinko tris kriterijus pagal kurias vertinami rezultatai – gramatika, tema ir bendras vertinimas. Rezultatuose pabrėžiama, kad straipsnio [FGD18] *MaskGAN* tinkle rezultatai gerokai lenkia tiek *LM*, tiek *MaskMLE* architektūras visose srityse. Tačiau žmonės vis dar ganėtinai lengvai atskyrė, kurie tekstai buvo sugeneruoti neuroninio tinklo, o kurie yra originalūs žmogaus kurti tekstai.

1.4.2.2. CS-GAN

Straipsnyje [LPW+18] pristatomas naujas karkasas – sakinius pagal kategorijas kuriantis generatyvinis besivaržantis neuroninis tinklas (angl. *category sentence-generative adversarial networks, CS-GAN*) (17 pav.). Šio tinklo architektūrą, kaip ir *MaskGAN*, sudaro rekurentiniai neuroniniai tinklai ir mokymosi su paskatinimu modelis. *CS-GAN* naudoja *LSTM* modelį. Kaip ir darbe [FGD18] autoriai pabrėžia, kad pagrindinė problema su kuria *GAN* susiduria generuodami tekstą yra diskreti jo prigimtis sunkiai perteikiama matematiškai. Mokymosi su skatinimu modelio pasirinkimą autoriai pagrindžia, kad natūralu yra leisti sprendimus priimti agentui (angl. *agent*), kuris atsižvelgia į tikslą (nurodytą kategoriją) ir remiasi turimais duomenimis. Taip siekiama imituoti žmogaus mąstymą.

Autoriai pabrėžia, kad dirbtinės realybės kūrimo užduotis yra sudėtinga remiantis tradiciniais neuroninių tinklų modeliais – apmokant su žinomais duomenų rinkiniais. Todėl šiame darbe jie orientuosis į teksto sakinių kūrimą pagal kategorijas duomenų rinkinių praplėtimui (angl. *data augmentation*). Šį uždavinį autoriai išskirsto kaip du pagrindinius iššūkius:

1. Generuoti žmogui suprantamus sakinius su *GAN*;
2. Pridėti kategoriją, pagal kurią *GAN* generuotų sakinius.



16 pav. *CS-GAN* struktūra. c - kategorijos informacija, z - įvesties duomenys, d_g - generatoriaus išvestis, s_r ir d_r reali žyma ir sakinys, brūkšininės linijos yra apribojimai

Teksto generavimas yra natūralios kalbos generavimo problema reprezentuojant turimą informaciją. Straipsnyje [LPW+18] ji sprendžiama generuojant sužymėtus sakinius, kurie leidžia paskirstyti išvesties reikšmes pagal kategorijas. Norint tai pasiekti neuroninis tinklas apmokomas dviem etapais: paduodami konkrečios kategorijos įvesties duomenys taip verčiant modelį generuoti tik šios kategorijos sakinius.

Straipsnio autoriai argumentuoja *RNN* pasirinkimą kaip geriausią modelį generuoti seką atsimenant prieš tai sukurto teksto kontekstą. Jie inkorporuoja šį tinklą kaip generatyvinį *GAN* tinklą. Diskriminatorius kiekvienoje iteracijoje gauna realius ir *GAN* sugeneruotus sakinius. Šis modelis veikia taip pat kaip standartinis *GAN* – varžybos tarp dviejų neuroninių tinklų, kur vieno klaida yra kito pergalė. Mokymosi su skatinimu modelio pasirinkimą autoriai pagrindžia, kaip ilgalaikę žaidimo strategiją. Agentas renkasi žetoną generavimui žiūrėdamas į ateitį, o diskriminatorius priešingai – siekia gerų rezultatų iš karto.

Straipsnio [LPW+18] rezultatai pateikiami 17 pav. Autoriai pasinaudojo *Amazon*, *Emotion* bei *News* duomenų rinkiniais bei testavo *CS-GAN* su mokymosi su skatinimu ir be. Geriausius rezultatus mažame duomenų rinkinyje pavyko pasiekti naudojantis *CS-GAN* su mokymosi su skatinimu, o didesniame duomenų rinkinyje *Amazon-30000* geriausiai pasirodė be *RL*.

Modelis	Amazon-5000	Amazon-30000
CNN	84.83%	89.55%
CS-GAN w/o RL&GAN	85.60%	**89.67%**
CS-GAN w/o RL	86.18%	89.54%
CS-GAN	**86.43%**	89.34%
Modelis	Emotion-15000	NEWS-15000
CNN	40.75%	72.08%
CS-GAN w/o RL&GAN	39.32%	72.31%
CS-GAN w/o RL	40.14%	72.09%
CS-GAN	**41.52%**	**74.33%**

17 pav. *CS-GAN* rezultatai ([LPW+18])

1.4.3. Abstraktaus sintaksės medžio generavimas su GAN

1.4.3.1. TreeGAN

Straipsnyje [LKL+18] pritaikomi generatyviniai besivaržantys neuroniniai tinklai sekos, kuri turi specifines gramatikos taisykles, generavimo problemai spręsti. Tokia seka galėtų būti programinis kodas. Neuroninį tinklą pavadino medžio *GAN* (angl. *TreeGAN*), kadangi autoriai, kaip ir [YN17; YN18] straipsniuose generuos abstrakčius sintaksės medžius.

Straipsnyje išskiriami keli pagrindiniai iššūkiai su kuriais susiduria *GAN* generuodami programinės kalbos kodą:

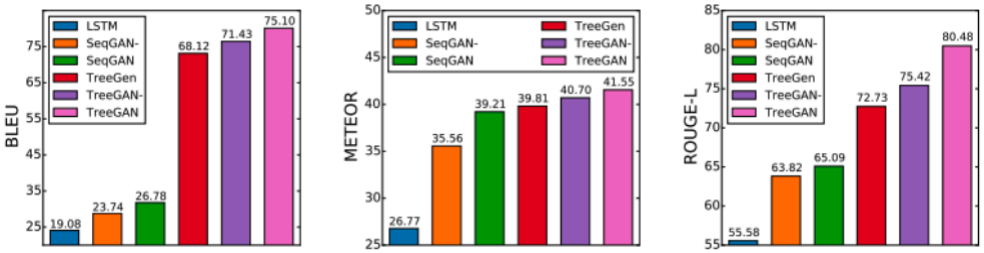
1. Užtikrinti sintaksės korektiškumą. Sugeneruotas kodas turi būti suprantamas kompiliatoriaus arba interpretatoriaus;
2. Diskriminatorius turi suprasti sintaksę, taip pat kaip ir generatyvusis tinklas;
3. Neišbaigtos frazės sekos išlaikymas. Kol tinklas generuoja kodo sakinį, pridėdamas naują žetoną turi išlaikyti loginę seką sugeneruotą prieš tai.

Išspręsti šioms problemoms vietoj kodo generavimo kaip natūralios kalbos sekų su specialiomis gramatikos taisyklėmis autoriai renkasi generuoti abstrakčius sintaksės medžius. Kiekvienas sugeneruotas medis atstovauja validžią seką, kuri išlaiko reikiamas gramatikos taisykles.

Autorių siūlomoje architektūroje generatorius naudoja *LSTM* neuroninį tinklą, kuris generuotų medžius. Diskriminatorius taip pat būtų *LSTM*, tačiau ne įprastas, o taip pat paremtas medžio struktūra, kad galėtų validuoti ne tik realistiškumą, bet ir sintaksę.

Panašiai kaip straipsniuose [YN17; YN18], čia autoriai į medžio generavimo problemą žvelgia kaip į seką veiksmų (angl. *actions*) kurios sukonstruotų medžio struktūrą. Straipsnyje [LKL+18] pristatomi du pagrindiniai veiksmai: kodo taisyklių (angl. *production rules*) generavimas bei galutinių žetonų (angl. *terminal tokens*) generavimas. Galutiniai žetonai priklauso bekontekščiam gramatikos žodynui (angl. *context free grammar, CFG*). Generavimo procesas veikia einant gilyn į medį iš kairės į dešinę. *TreeGAN* diskriminatorius yra specialus medžio struktūros *LSTM* tinklas veikiantis nuo apačios viršūnių eidamas link medžio šaknies (angl. *Child-Sum Tree-LSTM*). Prieš pradedant apmokyti generatyvinį tinklą vykdoma prieš-apmokymo fazė (angl. *pre-training*). Joje diskriminatorius apmokamas su dalimi gramatiškai teisingų medžių bei dalimi kur atsitiktinai parinktos medžio viršūnės pakeičiamos neteisingomis. Tai diskriminatoriui leidžia išmokti teisingos programavimo kalbos sintaksės.

Rezultatus autoriai pateikia lygindami savo kuriamo tinklo *TreeGAN* architektūrą su keliomis kitomis: *LSTM, SeqGAN, TreeGen*. Pasirinktos programavimo kalbos apmokyti tinklus buvo: *Python Django, SQL* bei *PLD*. *Django* rezultatai pateikiami 18 pav.



18 pav. *Django* karkaso testavimo rezultatai ([LKL+18])

1.5. Transformerių neuroniniai tinklai

1.5.1. Transformerių neuroninių tinklų architektūra

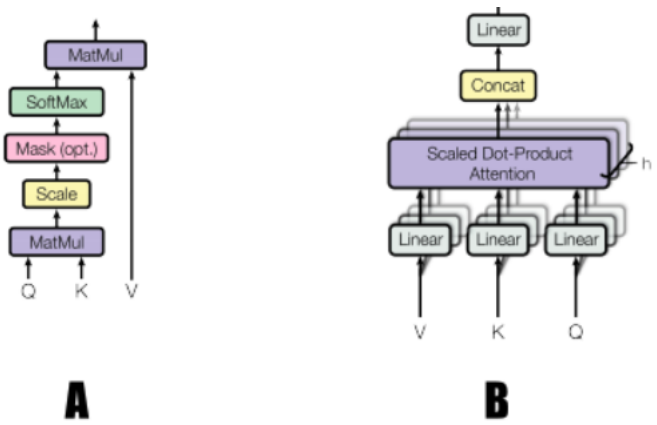
Transformerių neuroniniai tinklai pirmą kartą pristatyti straipsnyje [VSP+17]. Autoriai pristato naują neuroninių tinklų architektūrą spręsti sekų problemas – vertimus, natūralios kalbos generavimą. Šio straipsnio pavadinimas – dėmesys yra viskas ko reikia (angl. *Attention is all you need*) atitinka transformerių neuroninių tinklų architektūros pagrindinę idėją.

Transformerių neuroniniai tinklai yra kaip alternatyva rekurentiniams neuroniniams tinklams. Straipsnio autoriai išskiria greitesnį mokymo laiką, lygiagretinimo palaikymą, kitokį dėmesio mechanizmą (angl. *attention*) kaip pagrindinius transformerių neuroninių tinklų pranašumus prieš rekurentinius neuroninius tinklus. Transformerių neuroninių tinklų dėmesio mechanizmo lygtis [VSP+17]:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

- Q (angl. *query*) – vieno žodžio vektorius sekoje
- K (angl. *keys*) – visų žodžių raktų vektorius sekoje
- V (angl. *values*) – visų žodžių reikšmių vektorius sekoje
- d_k (angl. *dimensions*) – raktų dimensijų skaičius
- softmax – funkcija, kuri padaro, kad dėmėsio sutelkimo reikšmės būtų tarp 0 ir 1

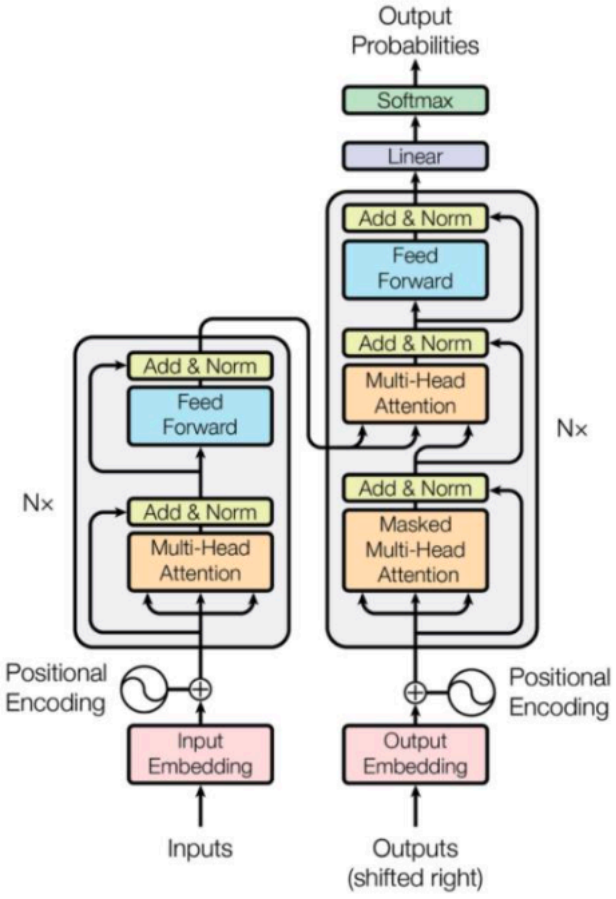
Ši lygtis transformerių neuroniniuose tinkluose gali būti pritaikoma tiek nuosekliai 19 pav. a, tiek lygiagrečiai 19 pav. b skaičiuojant *attention* reikšmę kiekvienai dimensijai atskirai.



19 pav. Transformerių neuroninių tinklų paralelizmo pavyzdys ([VSP+17])

Transformerių neuroninių tinklų architektūra sudaryta iš *encoder-decoder* tinklų. Šioje architektūroje transformerių tinklai naudojasi savo-dėmesio (angl. *self-attention*) mechanizmu, pilnai sujungtais *encoder-decoder* sluoksniais ir kiekviename žingsnyje modelis įvesčiai paduoda praėjusios išvesties rezultatus.

Transformerių tinklų architektūra pavaizduota 20 pav. kairėje pusėje *encoder* tinklas, dešinėje – *decoder*. Abu tinklai yra sudaryti iš skirtingų modelių, kuriuos galima dėti vienus ant kitų, kaip galima matyti N_x blokuose.



20 pav. Transformerių neuroninių tinklų modelio architektūra ([VSP+17])

1.5.2. Žetonų generavimas su transformerių neuroniniais tinklais

Straipsnyje [SSS20] pristatomas įrankis *IntelliCode* programavimo įrankiui (angl. *Integrated development environment, IDE*) paremtas debesų kompiuterijos sprendimu (angl. *cloud*). *IntelliCode Compose* – įrankis gebantis generuoti sintaksiškai teisingus kodo sakinius pagal kontekstą ar kategoriją. Įrankio trumpas apibūdinimas yra *C* kodo kontekstui naudojantis *V* programavimo kalbos žodyną sugeneruoti seką žetonų m_t, priklausomą nuo prieš tai esančių c_t žetonų.

Transformerių neuroniniam tinklui apmokyti straipsnio [SSS20] autoriai surinko 1,2 milijardo kodo eilučių. Jas sudaro *Python, C#, Javascript* ir *Typescript* programavimo kalbos. Kodo eilutės surinktos iš 52 tūkstančių geriausiai įvertintų *Github* repozitorijų kolektyviai turinčių 4,7 milijonus failų. Autoriai naudojo 70-30 santykį padalinti duomenis programavimo ir testavimo rinkiniams, o programavimo rinkinys dar buvo padalintas 80-20 proporcija apmokymo ir validavimo rinkiniams. Galutinis modelis įdiegtas su visais duomenimis.

Straipsnio [SSS20] transformerių neuroninio tinklo architektūrą sudaro lygiagretus *self- attention* blokas (19 pav. b) ir dviejų sluoksnių daugiasluoksnio perceptrono modelį.

Šiame darbe [SSS20] autoriai atsisako minties generuoti abstraktų ar konkretų (angl. *concrete syntax tree*) sintaksės medį, nes jų nuomone tai sulėtina paprastą kodo eilučių užbaigimo įrankio veikimą bei prideda papildomų nereikalingų priklausomybių. Taip pat pabrėžia, kad norint generuoti kodo eilutės užbaigimą kontekste esanti sintaksė gali būti netaisyklinga, todėl gali nepavykti sugeneruoti abstraktaus sintaksės medžio. Šis įrankis remiasi programinės kalbos gramatika bei aplink esančio kodo sintaksės medžiu.

Viena iš problemų su kuria susiduria straipsnio [SSS20] autoriai yra skirtingas tarpų tipų panaudojimas. Šiaip problemai išspręsti jie transformuoja kodą į žetonų seką naudodamiesi savo sukurtu pagalbiniu įrankiu. Prieš duomenų padavimo, apmokymo ir panaudojimo atvejais, duomenys yra apdirbami, sudedamas žetonų žodynas ir yra užkoduojamos sekos.

Kita didelė problema yra jautrios informacijos perdavimas įrankiui. Kadangi įrankiui apmokyti naudojamos viešos kodo saugyklos (*Github* repozitorijos), tai natūralu, kad programuotojai yra palikę slaptažodžių, raktų ar kitos jautrios informacijos. Šią informaciją reikia panaikinti, prieš apmokant neuroninį tinklą. Autoriai šiai problemai išspręsti sugeneruoja specialius žetonus komentarams, žodinės eilutės reikšmėms (angl. *string literals*). Pagal tai kaip dažnai panaudojami šie žetonai su konkrečiomis reikšmėmis autoriai išvalė duomenų rinkinį ir panaikino rečiausiai sutinkamas reikšmes. Pavyzdžiui iš *JavaScript* buvo atrinka 200 pasikartojančių žodžių eilučių, kurios sudaro net 20 % šios kalbos duomenų rinkinio.

Straipsnio [SSS20] autoriams pavyko sukurti vieną neuroninį tinklą visoms numatytoms programavimo kalboms. Tam jiems teko patobulinti duomenų rinkinius klasifikuojant juos prieš apmokymo fazę – programinės eilutės buvo papildytos žyma kokia programine kalba ji yra parašyta. Neuroninio tinklo architektūra pagal šią žymą atsirenka žetonų rinkinį ir taip geba generuoti programinio kodo eilutes skirtingoms kalboms.

1.5.3. Kodo generavimas su transformerių neuroniniais tinklais remiantis ASM

1.5.4. TreeGEN

TreeGEN – įrankis, paremtas medžio struktūra, pristatomas [SZX+20]. Šio neuroninio tinklo modelis primena *TreeGAN* pristatytą [LKL+18]. *TreeGEN* naudojasi transformerių neuroninių tinklų *attention* mechanizmu išspręsti ilgalaikių priklausomybių problemą bei naudoja ASM skaitytuvą *encoder* tinkle. Autoriai, norėdami išlaikyti kodo struktūrą naudodamiesi ASM, pabrėžia, kad kol kas nėra tokios transformerių neuroninių tinklų architektūros, kuri tai sugebėtų padaryti. Kaip sprendimą jie siūlo sukombinuoti kodo eilutės vektorius, kartu su aplink esančiomis medžio viršūnėmis papildomame tinklo sluoksnyje, tačiau pripažįsta, kad nėra aišku kaip tai padaryti.

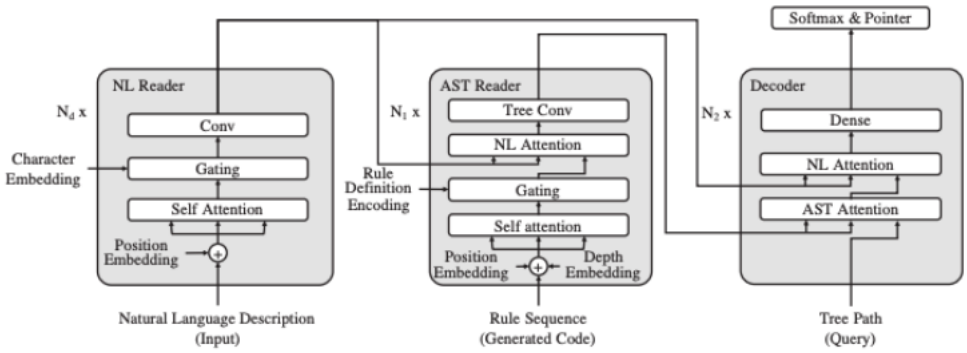
Straipsnio [SZX+20] autoriai minėtą sluoksnį planavo pridėti kiekviename transformerio tinklo bloke, tačiau norėdami, kad neuroninis tinklas išlaikytų konkrečios kodo eilutės viršūnės reprezentaciją per skirtingus sluoksnius, jie atsisakė šios minties – tai būtų pridėję papildomos nereikalingos informacijos šiai kodo viršūnei kiekviename sluoksnyje. Dėl šių priežasčių, autoriai savo tinklo architektūroje pridėjo sluoksnį gebanti išlaikyti medžio struktūrą į kelis pirmuosius decoder tinklo blokus.

Apibendrinus, *TreeGEN* architektūra susideda iš trijų pagrindinių dalių (21 pav.):

1. Natūralios kalbos įvesties nuskaitymo tinklas encoder, kuris užkoduoja įvestį;
2. ASM nuskaitantys blokai (keli pirmi *decoder* sluoksniai);
3. *Decoder*, likę sluoksniai sujungiantys viršūnę, kurią reikia papildyti ASM viduje bei naujai sugeneruotus žetonus pagal encoder tinklo įvestį.

ASM nuskaitymo blokai susideda iš keturių vidinių sluoksnių. Pirmasis – *self-attention* sluoksnis ištraukia informaciją iš ASM. Antrasis – sklendžių mechanizmas (angl. *gating mechanism*) sujungia turinio užkodavimo taisykles. Trečiasis – natūralios kalbos dėmesio sluoksnis (angl. *NL attention*), šiame sluoksnyje *decoder* gauna informaciją iš *encoder* apie įvestį. Paskutinis – medžio konvoliucija (angl. *tree convolution*), sujungia nagrinėjamą medžio viršūnę su tėvinėmis viršūnėmis – išlaiko medžio struktūrą.

Paskutinė architektūros dalis – *decoder* tinklas. Sujungia sugeneruoto kodo informaciją, natūralios kalbos įvestį iš *encoder* tinklo ir sugeneruoja naują gramatikos taisyklę, kurią įterpia į ASM viršūnę.



21 pav. *TreeGEN* architektūra ([SZX+20])

1.6. Skyriaus apibendrinimas ir išvados

1.6.1. Rekurentiniai neuroniniai tinklai

Šioje šaltinių analizėje buvo išnagrinėti kelios skirtingos *LSTM* architektūros bei aprašytas rekurentinių neuroninių tinklų veikimo principas. Nagrinėti straipsniai rinkosi *LSTM* tinklus, todėl, kad jie sprendžia dvi pagrindines *RNN* problemas – nykstančio gradiento bei ilgalaikių priklausomybių ilgose sekose. Analizėje apžvelgiamos *LSTM* galimybės generuoti programinės kalbos kodą naudojantis abstrakčiais sintaksės medžiais ir žetonais.

Visos nagrinėtos *LSTM* architektūros rėmėsi *encoder-decoder* neuroniniais tinklais. Straipsniuose [YN17; YN18] *LSTM* architektūra generuojanti abstrakčius sintaksės medžius rėmėsi veiksmų kūrimu kuriant po vieną medžio viršūnę. Tai leido pasiekti atitinkamai 71,6 % ir 73,7 % tikslumą generuojant *Python Django* kodą.

1.6.2. Generatyviniai besivaržantys neuroniniai tinklai

Analizėje taip pat nagrinėti generatyviniai besivaržantys neuroniniai tinklai bei jų pristatymas [GPM+14] „*Generative Adversarial Networks*“ straipsnyje. Šių neuroninių tinklų veikimo principas yra varžybos tarp dviejų tinklų – generatyvinio bei diskriminatoriaus. Generatyvinis tinklas iš pseudo-atsitiktinių duomenų bando sugeneruoti rezultatus, kurių diskriminatorius neatskirtų nuo realių.

Kadangi *GAN* paprasta architektūra nėra pritaikyta kodo generavimui pirmiausia apžvelgiamos dvi architektūros – *MaskGAN* ir *CS-GAN*. Abi šios architektūros naudojasi rekurentiniais neuroniniais tinklais bei mokymosi su skatinimu algoritmais generatyviniam tinklui. Tokie pasirinkimai yra atlikti todėl, kad *GAN* sunku generuoti tekstą dėl diskrečios jo prigimties – neįmanoma perduoti gradiento reikšmės iš diskriminatoriaus į generatorių. Naudojantis mokymosi su paskatinimu agentą, jis turi galimybę pats priimti sprendimus realioje aplinkoje. Abi teksto generavimo architektūros yra artimos analizuotoms *LSTM* architektūroms – *MaskGAN* naudojasi *encoder-decoder* tipo architektūra, o *CS-GAN – LSTM*.

Paskutinė analizuota *GAN* architektūra – *TreeGAN*. Ši architektūra generuoja kodą kuriant abstraktų sintaksės medį. Pagrindinės problemomis, su kuriomis susidūrė šios architektūros autoriai yra sintaksės korektiškumo užtikrinimas, diskriminatoriaus apmokymas suprasti sintaksę, neišbaigtos

frazės sekos išlaikymas. Čia autoriai kaip ir apžvelgtuose *LSTM* modeliuose generatyviniam tinklui naudoja *LSTM* modelį bei į medžio generavimą žvelgia kaip į veiksmų generavimo seką.

1.6.3. Transformerių neuroniniai tinklai

Transformerių neuroniniai tinklai – paskutiniai analizuoti tinklai. Pirmą kartą pristatyti 2017 metais straipsnyje „*Attention is all you need*". Šie neuroniniai tinklai yra kaip alternatyva rekurentiniams neuroniniams tinklams. Jų pagrindiniai pranašumai yra greitesnis mokymo laikas, paralelizmas bei dėmesio mechanizmas.

Analizėje ištirti du transformerių neuroninių tinklų modeliai – *IntelliCode compose*, kuris negeneruoja abstraktaus sintaksės medžio bei *TreeGEN* – kuris generuoja. *IntelliCode compose* įrankis yra kodo eilutės užbaigimo įrankis, todėl buvo pasirinkta nenaudoti abstraktaus sintaksės medžio. Turint nepilną kodo eilutę sudarytas medis gali būti nevalidus – gali trūkti viršūnių, kintamųjų deklaravimų. Šio modelio autoriai naudojo sintaksės žodyną bei žetonų generavimą. Tinklui apmokyti pasinaudojo netgi 1,2 milijardo viešai prieinamo kodo eilučių skirtingų programavimo kalbų.

TreeGEN architektūra vėl gi labai artima prieš tai apžvelgtų *LSTM* ASM generavimo modeliams. Čia naudojami *encoder-decoder* tipo neuroniniai tinklai. Autoriai į įprastą transformerių neuroninių tinklų architektūrą *decoder* tinkle į pirmus kelis sluoksnis įterpia specialų bloką gebantį išlaikyti medžio struktūrą.

1.6.4. Bendras palyginimas

Remiantis nagrinėtais šaltiniais galima daryti prielaidą, kad kodo generavimo srityje daugiausiai ištirti yra rekurentiniai neuroniniai tinklai. Tai yra natūralu, nes *GAN* (2014) bei transformerių neuroniniai tinklai (2017) pristatyti kur kas vėliau nei pavyzdžiui *LSTM* (1997).

LSTM tinklai, naudodami *encoder-decoder* struktūrą ir abstrakčius sintaksės medžius, ganėtinai tiksliai generavo *Python Django* kodą. Tačiau, nepaisant jų gebėjimo spręsti nykstančio gradiento ir ilgalaikių priklausomybių problemas, jie nėra optimalūs sprendimai dėl mažesnio efektyvumo nei transformeriai.

GAN architektūros, tokios kaip *MaskGAN* ir *CS-GAN*, taip pat naudoja rekurentinius neuroninius tinklus, bet jie susiduria su problemomis generuojant tekstą dėl diskrečios jo prigimties. *TreeGAN*, kuris generuoja kodą kuriant abstraktų sintaksės medį, susiduria su problemomis, užtikrinant sintaksės korektiškumą ir diskriminatoriaus apmokymą.

Transformerių neuroniniai tinklai, tokie kaip *IntelliCode Compose* ir *TreeGEN*, pasižymi greitesniu mokymo laiku, paralelizmu ir dėmesio mechanizmu ir tai suteikia jiems pranašumą prieš *LSTM* ir *GAN* modelius. *TreeGEN* naudoja specialų bloką, kuris išlaiko medžio struktūrą, todėl jis yra artimas *LSTM* ASM generavimo modeliams.

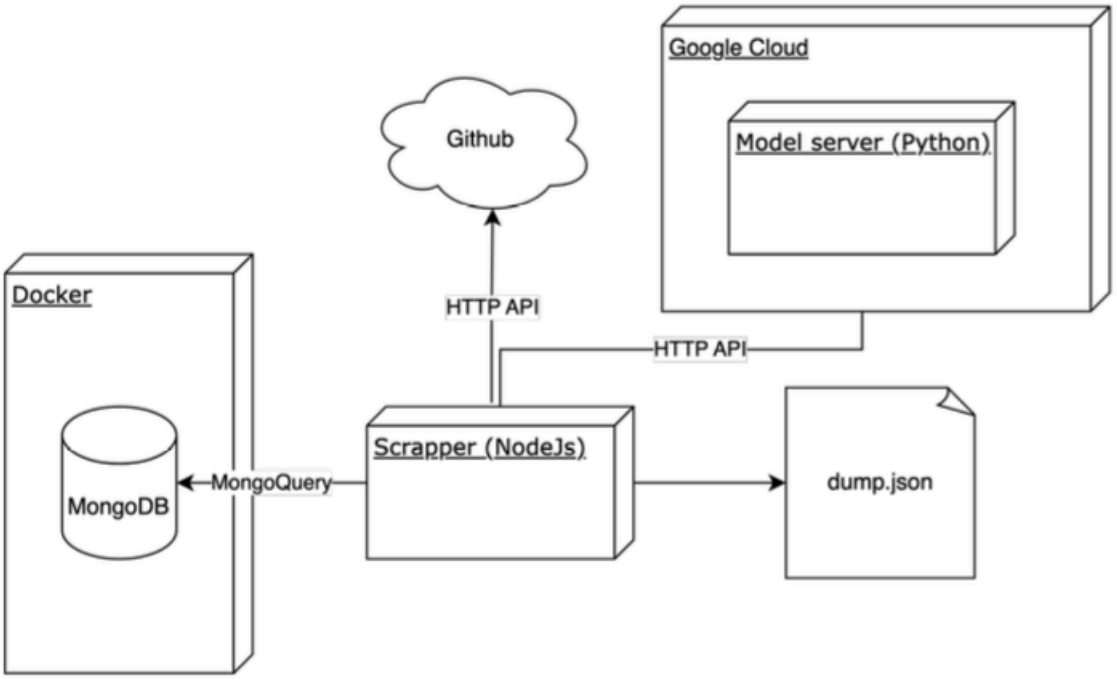
Remiantis šia analize, transformerių neuroninių tinklai atrodo kaip tinkamiausia architektūra šio darbo tikslams įgyvendinti. Transformerių neuroninių tinklų trumpesnis apmokymo laikas ir didesnis efektyvumas nei *LSTM* ar *GAN* architektūros. Todėl šiame magistro baigiamajame darbe

bus naudojami transformerių neuroniniai tinklai *JavaScript* kodo iš abstrakčių sintaksės medžių generavimui.

2. Modelio apmokymas

2.1. Duomenų surinkimas

Vienas iš darbo uždavinių yra duomenų surinkimas iš atvirojo kodo platformų. Pati didžiausia ir populiariausa pasaulyje atvirojo kodo platforma yra *Github* saugykla. 2018 metais *Github* pasiekė 100 milijonų kodo repozitorijų per 31 milijoną programuotojų iš viso atlikusių 1,1 milijardų kodo pasidalinimų (angl. *commit*). *Github* pateikia programos programavimo sąsają (angl. *application programming interface, API*), per kurią galima parsisiųsti bet kurio programuotojo viešas repozitorijas. Pagal *Github* licenciją, atviras kodas gali būti naudojamas nekomerciniais, edukaciniais tikslais. Saugykla riboja *API* užklausų kiekį siekiant apsisaugoti nuo robotų atakų, tačiau turint patvirtintą *Github* paskyrą galima atlikti iki 5000 užklausų per valandą iš vienos registracijos. Tam, kad būtų galima dirbti su įvairių programuotojų duomenimis, buvo sukurtas, kuris nuskaito *Github* repozitorijų duomenis - interneto griaužėjas (angl. *web scraper*) (22 pav.).



22 pav. Interneto griaužėjo sąsajos

2.2. Interneto griaužėjas

Interneto griaužėjas yra valdomas per komandinės eilutės sąsają (angl. *command line interface, cli*). Įrankis turi devynias komandas, kurios įgyvendina visą duomenų surinkimo ir paruošimo eigą - nuo kodo savininko pridėjimo iki galutinio failo ir rezultatų testavimo. Įrankio veikimo žingsniai pavaizduoti 23 pav.

23 pav. Įrankio veikimo žingsniai

2.2.1. Kodo savininko pridėjimas

CLI komanda: „npm run cli – –add-dev <devname>“. Ši komanda kreipiasi į *Github* API ir paima reikalingą programuotojo informaciją: *Github* naudotojo vardą, repozitorijų skaičių bei jų adresus. Komandoje esantis kintamasis <devname> yra unikalus ir naudojamas visose tolimesnėse komandose. Programuotojo informacija išsaugoma *MongoDB* „Developers“ kolekcijoje (34 pav.).

2.2.2. Repozitorijų pridėjimas

CLI komanda: „npm run cli – –refresh-repos <devname>“. Ši komanda kreipiasi į *Github API* ir rekursiškai partraukia visas programuotojo repozitorijas su reikalinga informacija: pavadinimas ir kokia programavimo kalba parašytas kodas. Programuotojo informacija išsaugoma *MongoDB* „Repos“ kolekcijoje (34 pav.).

2.2.3. Repozitorijų failų parsiuntimas

CLI komanda: „npm run cli – –index-repos <devname>“. Ši komanda taip pat kreipiasi į *Github API* ir kiekvienai programuotojo repozitorijai rekursiškai partraukia failus. Kai kurių failų tipas yra katalogas, todėl jo parsiųsti nereikia, bet reikia eiti gilyn ir parsiųsti visus jos viduje esančius failus ir taip toliau. Siunčiantis failus yra keletas taisyklių:

1. Ignoruojamos direktorijos: *node_modules* (saugo visų reikalingų bibliotekų kodą), *dist* arba *build* (saugo sukompiliuotą kodą). Pagal gerąsias programavimo praktikas nei viena iš šitų direktorijų turėtų neegzistuoti kodo saugykloje, tačiau neretai per klaidą žmonės jas ten išsaugo.

2. Failų pavadinimas bei dydis. Kartais, kliento dalies kode, yra išsaugomi minimizuoti *JavaScript* kalbos failai savyje turintys *Angular* karkasą (24 pav.) arba *React* biblioteką. Tokie failai dažnai yra kelių tūkstančių eilučių ilgio ir pavadinimas gali būti atpažintas su *Regex* formatu

   (angular.min|bundle).js ir panašiai.

3. Programavimo kalba. Nuskaitomi tik *JavaScript* arba *TypeScript* repozitorijų failai.

MongoDB „Files“ kolekcijoje išsaugomos failų parsisiuntimo nuorodos.

24 pav. Nenuskaitomo failo pavyzdys

2.2.4. Kodo eilučių klasifikavimas (taisyklių konfigūracija)

CLI komanda: „npm run cli – –parse-files <devname>“. Ši komanda parsisiunčia pačius kodo failus per *Github API*. Ši interneto griaužėjo dalis yra labiausiai kintanti siekiant kuo tiksliau apibūdinti kodo fragmentus ne tik individualiai, bet ir įtraukiant aplinkui esantį kontekstą. Taisyklės yra pritaikytos skirtingiems programavimo karkasams ir bibliotekoms. Šiuo metu tiksliausiai apibūdinamas yra *React* bibliotekos ir *Express* karkaso kodas. Keletas taisyklių pavyzdžių:

1. (const|let|var) - kintamojo deklaracijos eilutė
2. function .*
   (/.test(line) - funkcijos deklaracijos eilutė
3. line.includes(‚express‘) - *Express* karkaso failas ir kontekstas

```
73  function getReactLabels(line: string, ctx: ILineCtx): string[] {
74      const labels: string[] = [];
75      const nativeHooks = ['useState', 'useContext', 'useCallback', 'useMemo', 'useEffect'];
76
77      if (line.includes('import')) {
78          if (line.includes('react') || nativeHooks.some((nh) => line.includes(nh))) {
79              labels.push('react import')
80              labels.push('react')
81          }
82          if (line.includes('prop-types')) {
83              labels.push('react')
84          }
85          return labels;
86      }
87
```

25 pav. Konfigūruojami kodo fragmentų predikatai

Nuskaičius failą ir kiekvienai eilutei priskyrus jas apibūdinančias anotacijas (angl. *labels*), jos išsaugomos *MongoDB* „Lines“ kolekcijoje (34 pav.). Siekiant pagreitinti procesą pakeitus predikatus visi nuskaityti failai išsaugomi failų sistemoje, todėl ateityje norint iš naujo priskirti etiketes kodo eilutėms nereikės atlikti *Github API* užklausų.

2.2.5. Abstrakčių sintaksės medžių generavimas

CLI komanda: „npm run cli – –generate-ast <devname>“.

Ši komanda yra pati kompleksiškiausia ir yra skirta paruošti galutinius abstrakčius sintaksės medžius neuroninio tinklo apmokymui. Taip pat, komanda kai kurias sintaksės medžio viršūnes (o tiksliau – lapus, primityviąsias viršūnes) skaido į vienetus (angl. *character-based tokenization*). Tai reiškia, kad komanda iš sintaksės medžio formato paverčia į vieną trumpą eilutę, kuri vėliau gali būti atkurta atgal į sintaksės medį. Taip buvo nuspręsta daryti siekiant sumažinti duomenų rinkinio dydį ir supaprastinti mašininio mokymosi eigą.

Norint sugeneruoti *JavaScript* sintaksės medžius teko susidurti su dvejomis pagrindinėmis problemomis. Įrankis naudoja populiariausią sprendimą ASM generavimui: *Esprima* biblioteką. *Esprima* biblioteką naudoja tokie įrankiai kaip statinės kodo analizės įrankiai *eslint* ar *prettier* ar kodo dokumentacijos įrankis *Swagger*. Ši biblioteka nėra pritaikyta naudoti su *TypeScript* kalba, todėl teko nagrinėti kiekvieną individualią medžio viršūnę bei visoms joms aprašyti tipus. Įrankis iš viso apdoroja 57 skirtingus medžio viršūnių tipus.

Taip pat, *Esprima* yra skirta tik *JavaScript* kalbos abstrakčių sintaksės medžių generavimui. *TypeScript* kalbai beveik nėra bibliotekų tam atlikti, o egzistuojančios neturi jokios dokumentacijos. Todėl visi skaitomi *TypeScript* failai pirmiausiai buvo sukompiliuojami atgal į *JavaScript* kalbą, taip netenkant galimybės išsaugoti aprašytus kintamųjų tipus.

Detalūs komandos atliekami veiksmai:

1. Failo nuskaitymas.

   Naudojamas praeitame žingsnyje sukurtas kešas (angl. *cache*), tam, kad nereikėtų dar kartą kreiptis į *Github API*.

2. Failo atpažinimas ir ASM generavimas.

   Jei failas yra parašytas su *TypeScript* kalba jis pirmiausiai sukompiliuojamas atgal į *JavaScript* kalbą. Tada galima generuoti abstraktų sintaksės medį. *TypeScript* kompiliatorius leidžia rinktis kokį *EcmaScript* standartą turėtų įgyvendinti sukompiliuotas kodas, pasirinkus *ESNext* (patį naujausią standartą) sintaksė niekuo nesiskiria, tiesiog yra pašalinami programuotojo aprašyti tipai ir sąsajos (angl. *interfaces*) (26 pav.).

```
TS BaseMiddleware.ts > BaseMiddleware > add
1  import Wrap from '../abstract/Wrap';
2  import Ware from '../abstract/Ware';
3  import BaseMiddlewareHandler from '../abstract/BaseMiddlewareHandler';
4
5  export default class BaseMiddleware {
6      wares;
7      wraps;
8      constructor() {
9          this.wares = [];
10         this.wraps = [];
11     }
12     add(wareOrWrap) {
13         if (wareOrWrap instanceof Wrap) {
14             this.wraps.push(wareOrWrap);
15         }
16         if (wareOrWrap instanceof Ware) {
17             this.wares.push(wareOrWrap);
18         }
19         return this;
20     }
21 }
22
```

```
temp > aws-lambda-middleware > lib > abstract > TS BaseMiddleware.ts > ...
1  import Wrap from '../abstract/Wrap';
2  import Ware from '../abstract/Ware';
3  import BaseMiddlewareHandler from '../abstract/BaseMiddlewareHandler';
4
5
6  export default abstract class BaseMiddleware
7    <ReturnFunctionType, WrapFunctionType = unknown, WareFunctionArgs = unknown> {
8    protected wares: Array<Ware<WareFunctionArgs>>;
9    protected wraps: Array<Wrap<WrapFunctionType>>;
10
11   constructor() {
12     this.wares = [];
13     this.wraps = [];
14   }
15
16   add(wareOrWrap: BaseMiddlewareHandler): BaseMiddleware<
17     ReturnFunctionType,
18     WrapFunctionType,
19     WareFunctionArgs
20   > {
21     if (wareOrWrap instanceof Wrap) {
22       this.wraps.push(wareOrWrap);
23     }
24     if (wareOrWrap instanceof Ware) {
25       this.wares.push(wareOrWrap);
26     }
27     return this;
28   }
29
30   abstract getHandler(): ReturnFunctionType;
31 }
```

26 pav. *JavaScript* (kairėje) sukompiliuotas kodas iš *TypeScript* (dešinėje)

3. Rekursinis abstraktaus sintaksės medžio generavimas.

   Kiekvienas failas yra sudarytas iš vieno ar daugiau abstrakčių sintaksės medžių. Iš viso įrankis nuskaito 57 skirtingus medžių tipus, nuo smulkiausių (*Literal* ir *Identifier* iki kompleksiškiausių *CallExpression* ir *FunctionDeclaration*). Medžiai yra rekursiniai ir vienas tipas po savimi gali turėti keletą skirtingų šakų. 27 pav. pavaizduotas 8 eilučių kodo fragmentas sugeneruojantis 300 eilučių ilgio sintaksės medį. Medžio kompleksiškumą sudaro funkcija *requiresCallback*, kuri pirmu argumentu priima kitą funkciją, kuri taip pat priima argumentą. Visa tai yra aprašyta rekursinėje medžio struktūroje. Kiekviena individuali medžio šaka nuo pat funkcijos deklaracijos iki jos pavadinimo medžio lapo duomenų bazėje yra išsaugomi kaip atskiri abstraktūs sintaksės medžiai, nes kiekviena šaka individualiai yra validus kalbos kodo fragmentas. Patys smulkiausio kodo fragmentai *Literal* ir *Identifier* yra skirti aprašyti viską, kas nėra kalbos rezervuoti sintaksės žetonai (*function, class, const, throw, import*). Sakinyje „const myVariable = ‘example’" žodis *const* yra rezervuotas žetonas implikuojantis kintamojo deklaraciją ir medžio tipą *VariableDeclaration*. Kiekvienas kintamasis turi dvi šakas – *id*, kuri yra *Identifier* tipo ir *init*, kuri gali būti nuo elementaraus *Literal* tipo (šiuo atveju) iki pat funkcijos iškvietimo tipo *CallExpression*, kuris gali sugeneruoti dešimtis ar šimtus medžio šakų. Kintamojo *id* yra jo pavadinimas sukurtas programuotojo (28 pav. kairėje), šiame medyje tai yra „myVariable“. Kintamojo reikšmė yra *Literal* tipo (28 pav. dešinėje), šiame medyje tai yra „example“.

```
function requiresCallback(callback) {
  callback(1)
}

const object = {
  sum: requiresCallback((amount) => {
    return amount
  })
}
```

27 pav. Kompleksišką sintaksės medį turintis kodas

28 pav. Kintamojo pavadinimo (kairėje) ir reikšmės (dešinėje) medžio lapai

Kiekvienas medis turi „loc“ objektą, kuris nurodo nuo kokios eilutės prasideda kodo fragmentas ir kurioje baigiasi. Pagal tai galima identifikuoti konkrečias kodo eilutes originaliame faile ir iš duomenų bazės „Lines“ kolekcijos pasiimti visas šiam medžiui priklausančias anotacijas.

4. Medžio šakų tokenizavimas.

   Kodo sakinys „const myVariable = ‘example’“ sugeneruoja beveik 100 eilučių medį. Tai yra elementarus kintamojo deklaravimas, nenaudojantis objekto tipo, funkcijos kvietimo, kintamasis nepriklauso klasei ir nėra importuotas iš kito modulio ar bibliotekos. Visų šakų pabaigoje yra sutinkami primityvieji tipai *Literal* ir *Identifier*, kurie kartu užima apie 30 kodo eilučių. Šiame konkrečiame pavyzdyje tai sudaro 30 % medžio ir gali būti supaprastinti tiek dėl paprastesnio mašininio mokymosi algoritmo, tiek dėl duomenų failo dydžio. Taigi šiuos (ir dar kelius paprastus tipus) interneto griaužėjas išskaito į vienetus.

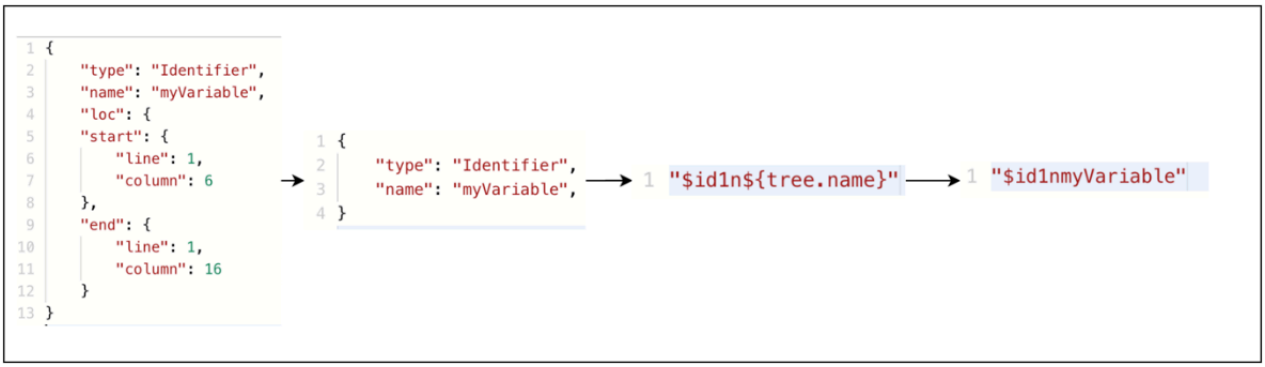
   Literal tipo tokenizacija pavaizduota 29 pav.

Šis tipas yra sudarytas iš apdirbtos reikšmės ir neapdirbtos programuotojo įvestos reikšmės. Norint atkurti pilnavertį sintaksės medį užtenka turėti programuotojo įvestą reikšmę ir priskirti žetoną, kad būtų galima identifikuoti tipą ir atkurti rezultatą.

29 pav. *Literal* tipo žetono sukūrimas

Identifier tipo tokenizacija 30 pav.

Šis tipas yra dar paprastesnis ir sudarytas tik iš programuotojo įvestos reikšmės. Norint atkurti pilnavertį sintaksės medį užtenka turėti šią reikšmę ir priskirti kitokį tipą identifikuojantį žetoną.



30 pav. *Identifier* tipo žetono sukūrimas

Taip pat, į vienetus skaidomi dar keli kiti paprasti medžiai: *ImportDefaultSpecifier*, *ExportAllDeclaration*, *TemplateElement*.

31 pav. Originalus sintaksės medis (kairėje) ir interneto griaužėjo rezultatas (dešinėje)

31 pav. galima matyti, kaip originalus kodo (32 pav.) medis yra 588 (kairėje) eilučių ilgio, o galutinis rezultatas 73 (dešinėje) ir su viršūnėmis pakeistomis žetonais. Panaudotas kodo pavyzdys yra paprasčiausias *Express* karkaso *HTTP* serverio paruošimas ir paleidimas.

```
import express from 'express'

import routes from './routes'

const app = express()

app.use('/api', routes)

export default (port) => {
    app.listen(port, () => console.info(`app listening on port ${port}`))
}
```

32 pav. *Express* kodo pavyzdys

5. Medžio išsaugojimas duombazėje.

   Medis duomenų bazėje išsaugomas kartu su jį apibūdinančiomis etiketėmis. Taip pat, prieš išsaugant medį yra ištrinami visi *loc* objektai, nurodantys nuo kurios iki kurios eilutės kodą apibūdina konkretus medis. Medžio sukūrimui *loc* objektas yra nereikalingas. Tai sudaro apie 50% originalaus medžio. Nagrinėtame kodo pavyzdyje su kintamojo deklaracija iš pradinio 80 kodo eilučių po žetonų generavimo ir *loc* objekto panaikinimo liko 17 kodo eilučių. Paskutinis optimizacijos žingsnis yra pakeisti medį iš *JSON* formato į *YAML*. Pastarasis yra paprastesnis ir trumpesnis. Šiame kodo pavyzdyje iš paskutinių 17 eilučių liko 11. Tai yra daugiau kaip 85% originalaus medžio sumažinimo. Galutinis sintaksės medis pavaizduotas 33 pav. Medis išsaugomas *MongoDB* „ASTs“ kolekcijoje (34 pav.).

```
type: Program
body:
- type: VariableDeclaration
  declarations:
  - type: VariableDeclarator
    id: '$id1nmyVariable'
    init: '$li2v"example"'
kind: const
sourceType: script
```

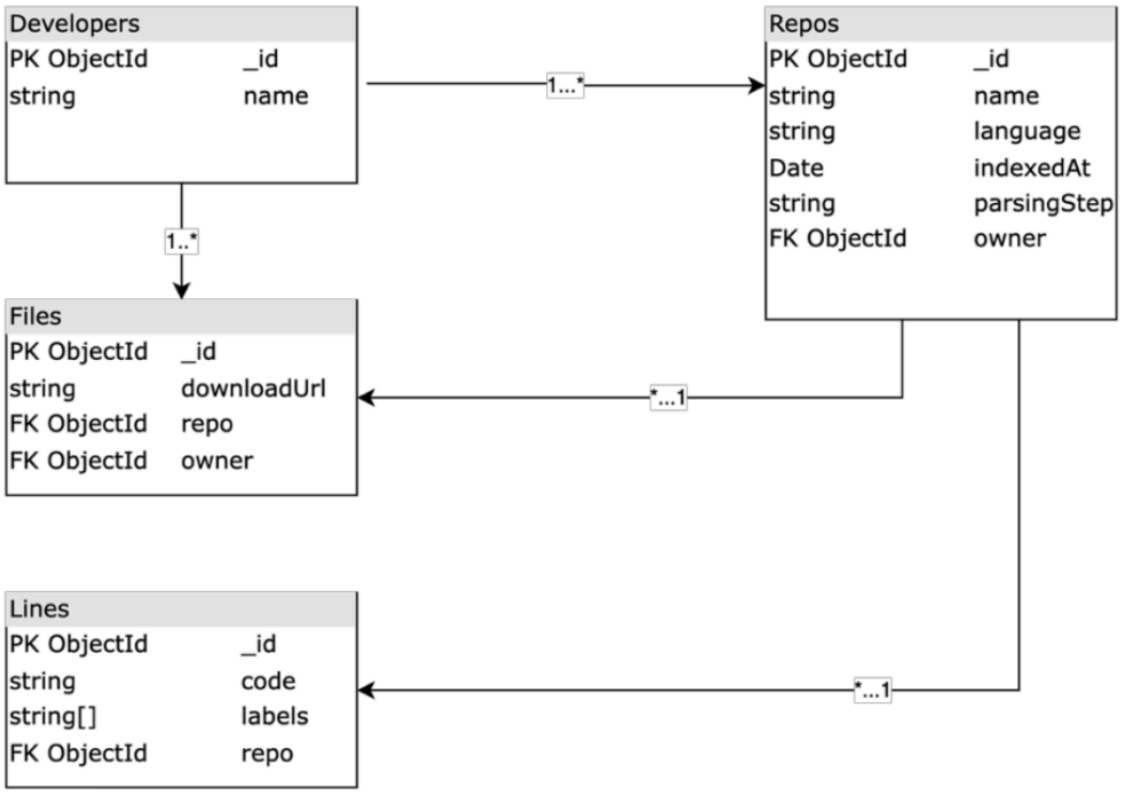
33 pav. Sintaksės medis *YAML* formatu po tokenizacijos ir optimizacijos

2.2.6. Duomenų failo sukūrimas

CLI komanda: „npm run cli – –dump-file“.

Ši komanda taip pat leidžia nurodyti keletą sąlygų, kokius sintaksės medžius norime matyti galutiniame faile. Pavyzdžiui galima nurodyti, kad į failą sugeneruotų tik 1000 sintaksės medžių,

kurių tipai yra *Literal* ir *Identifier*. Tai yra naudinga, norint apmokyti neuroninį tinklą keliais etapais. 1000 medžių sudarytų vien tik *Literal* ir *Identifier* tipo užima 94kb atminties, kompleksiškesni tipai (*FunctionDeclaration, ObjectExpression*) 1000 medžių užima 1,5mb.



34 pav. Duomenų bazės diagrama

2.2.7. Rezultatų testavimas

CLI komanda: „npm run cli – –validate-data“.

Ši komanda skirta testuoti apmokytą neuroninį tinklą naudojantis tiek apmokymo duomenimis, tiek testavimui paliktais duomenimis. Komanda iš nurodyto failo paima n=100 duomenų ir su jais kreipiasi į *Python Flask* serverį, kuriame įdiegtas apmokytas modelis. Modelis sugeneruoja rezultatą ir įrankis jį palygina su tikruoju kodu. Modelio tikslumas yra tikrinamas pagal kelis parametrus:

1. Levenšteino atstumas[3]. Šis matavimas pasirinktas todėl, kad parodo kiek modelio sugeneruotas rezultatas yra nutolęs nuo testuojamos įvesties, matavimas tiesiogiai koreliuoja su modelio kokybe.

2. *YAML* duomenų struktūros validumas. Įrankis patikrina ar sugeneruotas rezultatas atitinka *YAML* duomenų standarto semantiką. Nepavykus sugeneruoti semantiškai teisingo YAML formato, sugeneruoto rezultato neįmanoma paversti į kodą.

3. Medžio integralumas. Medžio šakų tipai turi atitikti *JavaScript* ASM taisykles, kitaip iš medžio nebus įmanoma atkurti kodo.

[3]Levenšteino atstumas nurodo kiek simbolių turi būti pakeista siekiant iš vieno teksto pereiti į kitą

2.2.8. Kodo eilučių klasifikavimas su OpenAI modeliu

CLI komanda: „npm run cli – –gpt-label“.

Ši komanda atrenka 15 tūkstančių svarbiausių tipų medžių: *VariableDeclaration, IfStatement, FunctionDeclaration* ir jiems su *OpenAI GPT-3.5-turbo* modeliu sugeneruoja prasmingas, žmogui suprantamas kodo anotacijas. Komanda siunčia įvesties užklausą modeliui per *API*, kuri atrodo taip:

Create a label for the following code: {"$code"} and do not exceed 15 words (lit.

Sukurk anotaciją šiam kodui: {"$kodas"} ir neviršyk 15 žodžių limito).

2.2.9. Įrankio naudojimas

CLI komanda: „npm run cli – –model-generate <input>“.

Ši komanda priima įvestį, kuri yra nusiunčiama per *API* į *Python* serverį (22 pav.), jame esantis modelis sugeneruoja rezultatus įvesčiai ir atsiunčia atgal interneto griaužėjui. Įrankis transformuoja *YAML* duomenų struktūrą į *JSON*, iš medyje esančių žetonų atkuria pirminius medžius ir su ASM bibliotekos pagalba iš turimo medžio sugeneruoja kodo fragmentą, kuris yra atspausdinamas į komandinę eilutę.

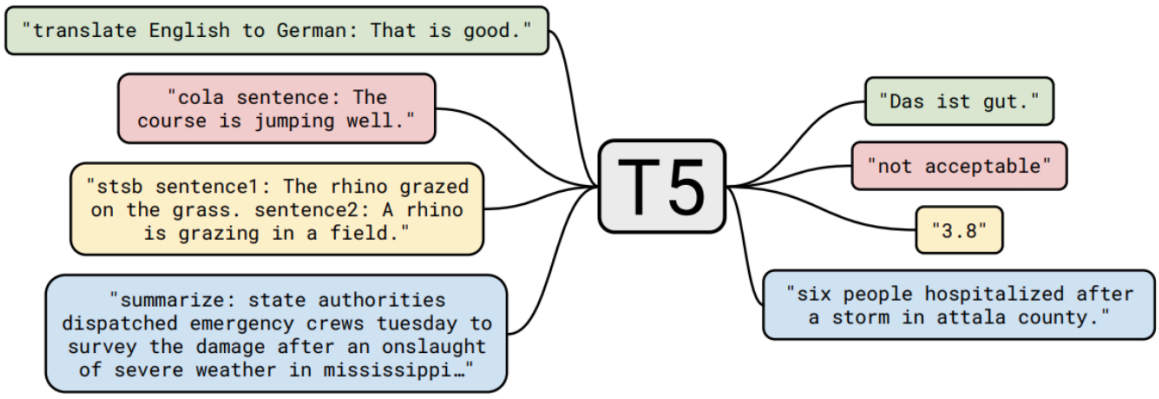
2.3. Surinktų duomenų rezultatai

Duomenims surinkti buvo naudojamos trijų programuotojų repozitorijos. Darbo autoriaus 8 asmeninės repozitorijos (30 tūkst. kodo eilučių) ir dviejų populiarių *YouTube* platformos kūrėjų, kurie užsiima *JavaScript* kalbos mokymu ir populiarinimu, daugiau kaip 200 repozitorijų (500 tūkst. kodo eilučių). Iš viso yra surinkta beveik 600 tūkst. abstrakčių sintaksės medžių. Nuskaityti, klasifikuoti, sugeneruoti, sukurti žetonai tiek medžių bei juos padėti į failą įrankiui užtrunka iki 30 minučių medžius generuojant sinchroniškai, tačiau interneto griaužėjas turi funkcionalumą lygiagrečiai dirbti su individualiomis repozitorijomis skirtinguose procesuose, todėl galutiniam duomenų surinkimui buvo naudojami trys atskiri procesai, kurie truko maždaug po 8 minutes ir kiekvienas paruošė beveik po 200 tūkst. medžių.

2.4. Modelis

2.4.1. Google T5 modelis

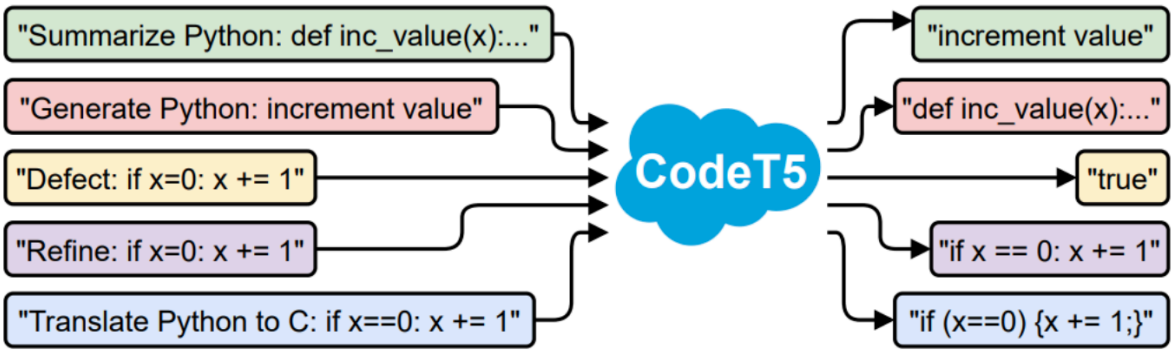
Google T5 algoritmas, aprašytas straipsnyje [RSR+20], yra transformerių tipo architektūros modelis sukurtas vieno teksto keitimo į kito teksto užduotims atlikti. Modelis naudojamas atlikti vertimams (pavyzdžiui iš anglų kalbos į vokiečių kalbą) ar sakinių apibendrinimui. Šis modelis yra pirmasis algoritmas, kuris bando aprėpti visas su tekstu susijusias užduotis į vieną formatą (35 pav.). *T5* pavadinime yra užšifruotas pats modelis ir jo atliekamas darbas: „*Test-to-Text Transfer Transformer*“ (lit. teksto į tekstą pakeitimo transformeris).



35 pav. *T5* modelio galimybių pavyzdys ([RSR+20])

T5 modelis siekdamas atskirti skirtingas užduotis ir jų užklausas naudojasi priešdėliais. (angl. *prefix*). Pavyzdžiui norint išversti vieną tekstą į kitą naudojamas priešdėlis „išversti iš anglų į prancūzų: “ (angl. *„translate English to French: “*). Priešdėliai gali būti naudingi siekiant atskirti įvairius kodo fragmentus vienus nuo kitų, pavyzdžiui funkcijos deklaraciją (*„Write a function: “*), kintamojo sukūrimą (*„Declare variable: “*), ar naujo klasės objekto iniciavimą (*„Create new: “*).

2021 Rugsėjį išleistame straipsnyje [WWJ+21] rašoma apie apmokytų algoritmų kaip *Google BERT* ir *GPT* panaudojimą kodo generavimo srityje. Jame aprašomas *CodeT5* algoritmas, sukurtas įmonės *Salesforce*, kuris yra pirmasis kodo sekas atpažįstantis *encoder-decoder* modelis. Jis yra apmokytas naudojantis *Google T5* 35 pav. modeliu aprašytu straipsnyje. *CodeT5* ne tik geba generuoti kodą (angl. *„Generate Python: “*), bet ir aprašyti jau sukurtą kodą (angl. *„Summarize Python: “*) ar netgi įvertinti paduotą sąlygą (angl. *„Defect: “*) (36 pav.).



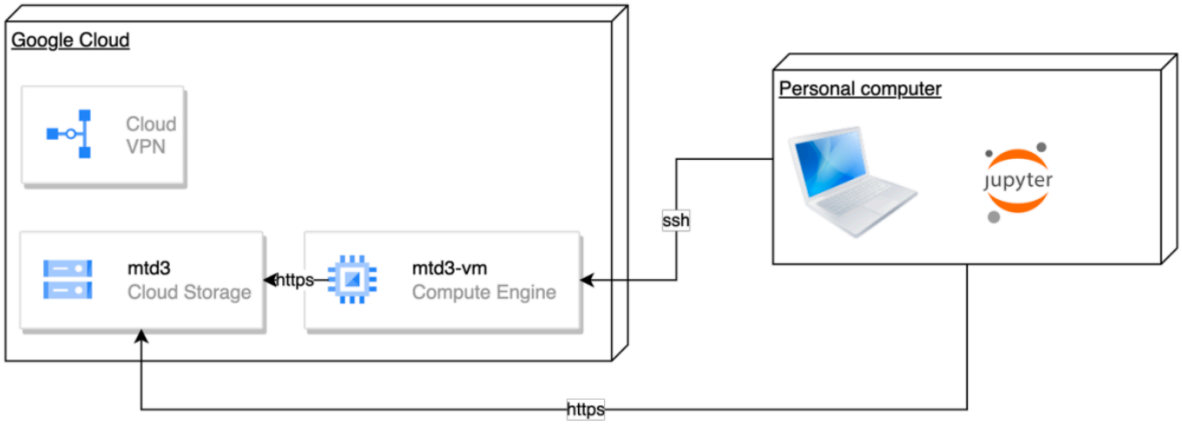
36 pav. *CodeT5* modelio galimybių pavyzdys ([WWJ+21])

CodeT5 modelis naudojasi *CodeSearchNet* duomenų rinkiniu, kurį sudaro tiek *labeled* duomenys tiek *unlabeled*. Modeliui apmokyti bus naudojami prieš tai minėti iš kodo sugeneruoti raktiniai žodžiai, tačiau viena iš alternatyvų klasifikuoti turimus duomenis būtų naudoti *CodeT5* modelį vietoj *OpenAI GPT* modelio.

Šiame darbe aprašomas modelis bus apmokomas naudojantis *Google T5* apmokytu neuroninio tinklo modeliu, kuriam bus pritaikytas *JavaScript* abstrakčių sintaksės medžių duomenų rinkinys. Modelis bus apmokomas skirtingais duomenų rinkiniais bandant išmokyti modelį ASM struktūros dalimis.

2.4.2. Infrastruktūra

Neuroninio tinklo apmokymui buvo panaudota *Google Cloud* 37 pav. debesų kompiuterijos sistema. Programavimo darbams atlikti buvo panaudota *Jupyter Notebook* sistema, dalis apmokymų buvo atliekami *Google Colab* platformoje. Duomenų rinkiniai ir apmokyti tarpiniai modeliai buvo saugojami *Google Cloud Storage* service, o pagrindiniam apmokymui naudotas *Google Cloud Compute Engine* serveriu. Modeliui apmokyti buvo naudojama *CPU* architektūra, nepagreitinta su *GPU*.



37 pav. *Google Cloud* infrastruktūra

2.4.3. Parametrai

Modeliui apmokyti buvo naudojama *Python* programavimo kalba ir *Tensorflow* biblioteka. Transformerių modelis buvo paimtas iš *transformers* bibliotekos, kuri naudoja *Hub5* neuroninių tinklų repozitoriją. Taip pat, iš šios bibliotekos buvo paimtas *tokenizatorius*, skirtas konvertuoti įvesties tekstą pritaikytą *Google T5* algoritmui.

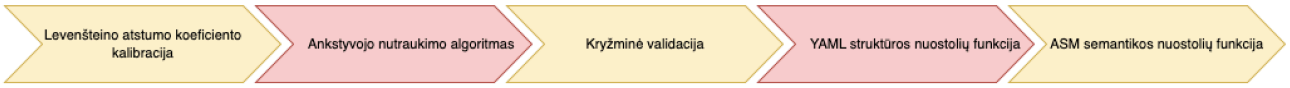
Duomenų rinkinio įvesčiai (*text*) ir kodui (*code*) buvo priskiriami žetonai naudojantis *RobertaTokenizer*, *Google T5* modelio apmokytu tokenizatoriumi. Tokenizatoriui taip pat reikia nurodyti, koks maksimalus leidžiamas įvesties ilgis. Priklausomai nuo duomenų rinkinio, ilgis gali skirtis nuo 10-20 simbolių (*Literal* ir *Identifier* tipo šakos), iki kelių šimtų ar tūkstančių (kompleksiškesni moduliai, funkcijos).

Suvienodinti duomenis, nepaisant skirtingų jų ilgių buvo naudojamos dėmesio užmaskavimai (angl. *attention masks*) su *Tensorflow from_generator* funkcija. Mokymuisi nenaudojamas *auto_shard_policy*, todėl, kad mokymosi algoritmas nebuvo padalintas per kelis procesus. Duomenys pritaikomi *Tensorflow* modeliams su funkcija *with_options*.

Naudotas *TFT5ForConditionalGeneration* modelis apmokytas *Google T5*. Modelis apmokomas epochomis, kiekvienoje epochoje yra pereinami visi duomenų rinkinio duomenys. Iteracijos pabaigoje atliekama validacijos funkcija, kuri skaičiuoja kokio dydžio nuostolis yra atliekant generavimą/prognozes. Pagal tai yra atnaujinami neuroninio tinklo svoriai. Modelis naudoja *reLU* aktyvacijos funkciją.

2.4.4. Strategija

Siekiant parinkti geriausius parametrus atsižvelgiant į turimus duomenis ir jų struktūrą buvo atliekami keli bandymai keičiant modelio parametrus. Pirmiausia modeliui apmokyti buvo naudojamos 5 epochos, vėliau pridėtas ankstyvojo sustojimo (angl. *early stopping*) mechanizmas atstatantis geriausius modelio svoris. Tobulinant modulį buvo keičiami validacijos funkcijos koeficientai (Levenšteino atstumas, *YAML* duomenų struktūra, ASM semantika). Modelio parametrų parinkimas buvo atliekamas su primityviomis viršūnėmis *Literal, Identifier*, o vėliau pridėjus *YAML* duomenų struktūros medžius validacijos funkcija tobulinta su *Property* ir *VariableDeclarator* medžiais. Atlikto darbo žigsniai pavaizduoti 38 pav. bei aptariami toliau.



38 pav. Modelio parametrų kalibravimo veiksmų seka

1. I apmokymų iteracija: *Identifier* tipas. Parametrai: 5 epochos, nuostolių funkcija: Levenšteino atstumo (1 lentelė) ir kryžminės entropijos sandauga. Duomenų rinkinio dydis: 1000.

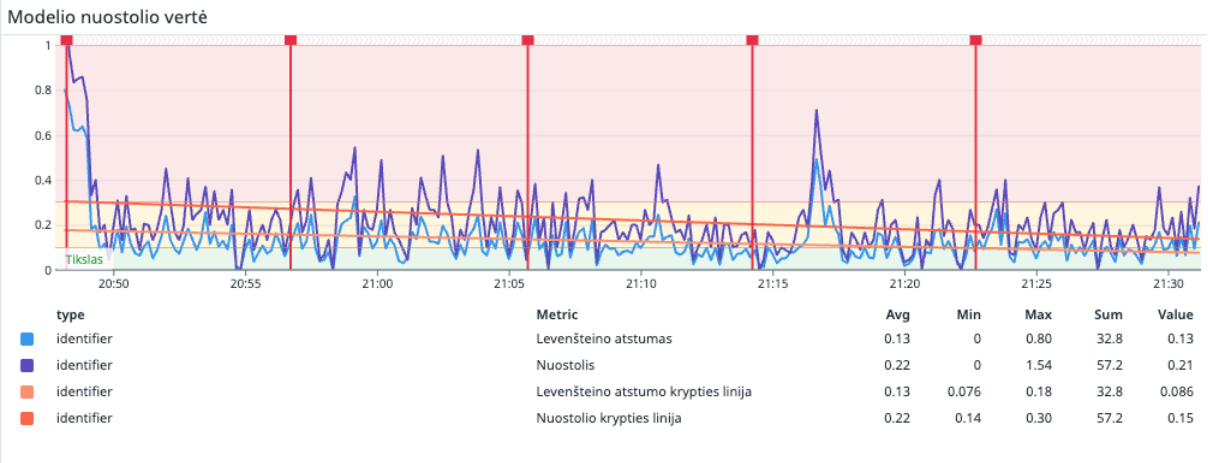
1 lentelė. Levenšteino atstumo koeficientai (I iteracija)

Atstumas (x %)	Nuostolis
x<=10	0
x<=30	0,4
x<=60	0,8
60<x	0,9

I iteracijos nuostolio vertės rezultatai ir koreliacija su Levenšteino atstumo rezultatais pavaizduota 39 pav. Jame matosi, kad paskutinėse dviejose epochose modelio rezultatai suprastėjo. Tai gali nutikti dėl persimokymo esant per mažam duomenų kiekiui ir per griežtai nustatytoms taisyklėms, nuostolio reikšmės per epochą pavaizduotos 2 lentelėje. Todėl tolimesnėje iteracijoje nuspręsta padvigubinti duomenų kiekį, pridėti ankstyvojo sustojimo algoritmą bei padidinti epochų kiekį. Žemiausia nuostolio vertė epochai buvo pasiekta trečioje epochoje - 0,1007, beveik siekianti norimą vertę - < 0,1.

2 lentelė. Nuostolio reikšmės per epochą (I iteracija)

Epocha	Nuostolis
1	0,3285
2	0,3133
3	0,1007
4	0,3352
5	0,4552



39 pav. I iteracijos modelio nuostolio vertė

2. II apmokymų iteracija: *Identifier* tipas. Parametrai: 8 epochos, nuostolių funkcija: Levenšteino atstumo (3 lentelė) bei kryžminės entropijos sandauga. Duomenų rinkinio dydis: 2000.

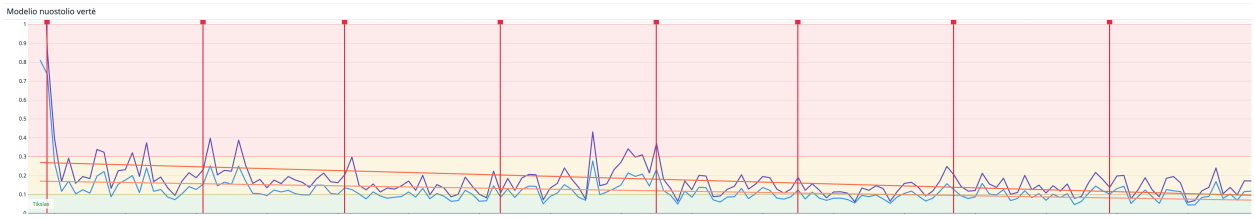
3 lentelė. Levenšteino atstumo koeficientai (II iteracija)

Atstumas (x %)	Nuostolis
x<=10	0
x<=20	0,1
x<=30	0,4
x<=40	0,6
60<x	0,8

Modelyje panaudota ankstyvojo stabdymo funkcija su 5 epochų tolerancija. Tai reiškia, kad modeliui netobulėjant penkias epochas iš eilės, bus nutrauktas modelio mokymasis ir bus atstatyti geriausi modelio svoriai. Žemiausia nuostolio vertė epochai buvo pasiekta penktoje epochoje - 0,0058, rodanti gerą modelio rezultatą (2 lentelė). Vėlesnėse epochose modelio rezultatai suprastėjo ir nuostolio vertė pakilo iki 0,0409, tačiau ankstyvojo nutraukimo algoritmas atstatė penktos epochos svorius. Šie rezultatai pateikti 40 pav. su 2000 duomenų yra neblogi, nors vėlesnėse epochose galima matyti persimokymo užuomazgų.

4 lentelė. Nuostolio reikšmės per epochą (II iteracija)

Epocha	Nuostolis
1	0,3285
2	0,1272
3	0,0207
4	0,0152
5	0,0058
6	0,0073
7	0,0209
8	0,0409



40 pav. II iteracijos modelio nuostolio vertė

3. III apmokymų iteracija: *Identifier, Literal* tipai. Parametrai: 5 epochos, nuostolių funkcija: Levenšteino atstumo (3 lentelė) bei kryžminės entropijos sandauga. Duomenų rinkinio dydis: 3000. Validacijos algoritmas: k-dalių kryžminė patikra (angl. *k-folds cross-entropy*).

5 lentelė. Kryžminės patikros rezultatai (III iteracija)

K-dalis	Geriausia epocha	Nuostolis
1	3/5	0,0547
2	3/5	0,0758
3	4/5	0,1347
4	4/5	0,0100

Kaip galima matyti 5 lentelėje, kryžminės patikros rezultatai yra ganėtinai skirtingi mokantis ant skirtingų duomenų rinkinio k-dalių. Tai gali rodyti modelio persimokymo problemą, o tai gali būti dėl per daug generalizuotos nuostolių funkcijos. Siekiant išvengti persimokymo sudėtingos nuostolių funkcijos skaičiavimai bus atliekami kas n=5 iteraciją. Tai taip pat pagerins modelio mokymosi greitį, ypatingai tolimesnėse iteracijose kai modelis bus apmokomas sudėtingesniais medžių tipais. Modelis buvo apmokytas su visu duomenų rinkiniu atlikus pakeitimus ir gauti svoriai naudoti tolimesniuose iteracijose.

4. IV apmokymų iteracija: įvairūs sudėtingesni tipai. Parametrai: 5 epochos, nuostolių funkcija: Levenšteino atstumo (3 lentelė), *YAML* duomenų struktūros validumo, medžio šakų semantikos bei kryžminės entropijos sandauga. Duomenų rinkinio dydis: 3000.

   Atrinkti duomenys turintys ne didesnį kaip 9 lygių gylį (siekiant supaprastinti duomenų rinkinio sudėtingumą) ir modelio nuostolių funkcija, modeliui sugeneravus *YAML* duomenų struktūrą teisingai, patikrins rezultato medžio šakų semantiką 3 lygių gylyje, o gilesnėms šakoms tikrins Levenšteino atstumą. Modelis taip pat bus baudžiamas sugeneravus nevalidžią *YAML* duomenų struktūrą, taip pat tokioms šakoms bus skaičiuojamas Levenšteino atstumo koeficientas.

   Rezultatai matomi 6 lentelėje, nors nuostolio reikšmė ir didesnė nei prieš tai vykdytose iteracijose, taip yra todėl, kad duomenų struktūra yra žymiai sudėtingesnė bei patys duomenys yra ilgesni (pavyzdžiui *Literal* tipas vidutiniškai yra iki 50 simbolių ilgio, o *Property* bent 400). Šioje parametrų kalibravimo stadijoje tikslinga stebėti *YAML* struktūros validumą (41 pav.). Nors sugeneruoti validūs *YAML* nesiekia 50 %, tačiau patikrinus kelis nesėkmingus atvejus matosi, kad pati medžio semantika teisinga, tiesiog yra kabučių neatitikimai. Mode-

6 lentelė. Nuostolio reikšmės per epochą (IV iteracija)

Epocha	Nuostolis
1	1,1620
2	0,9837
3	1,2312
4	0,3414
5	0,3862

lis buvo apmokytas vos su 3000 *YAML* medžių, todėl tokie rezultatai parametrų kalibravimo iteracijoje yra patenkinami.

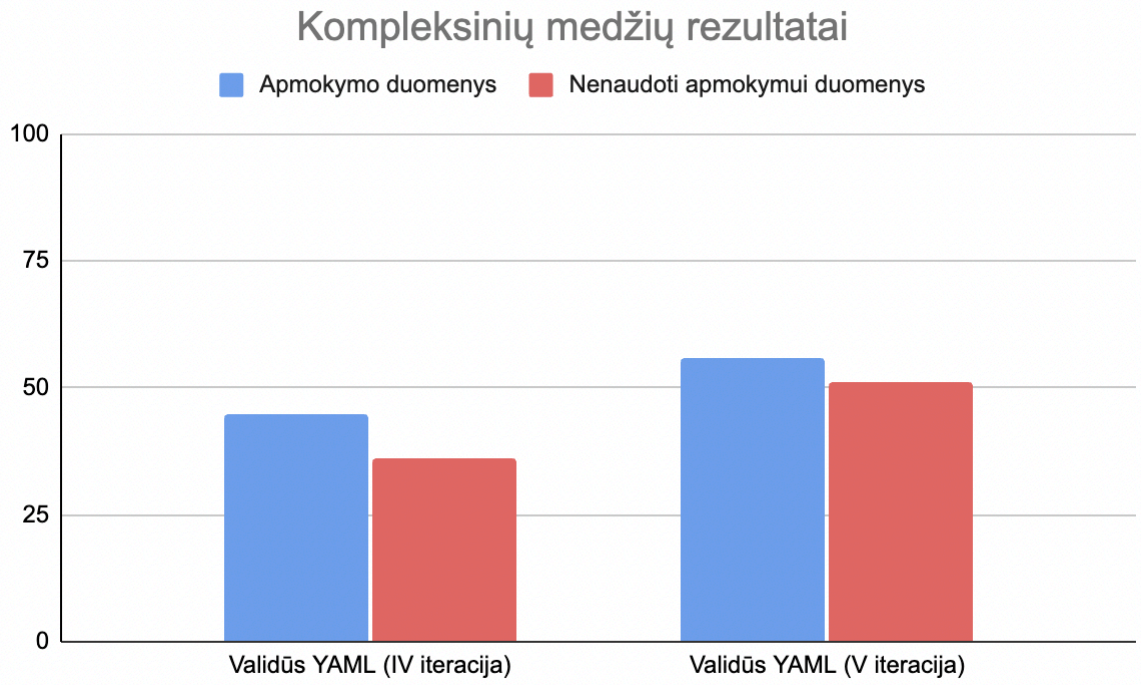
5. V apmokymų iteracija: *Property, VariableDeclaration* tipai. Parametrai: 10 epochų, nuostolių funkcija: Levenšteino atstumo (3 lentelė), *YAML* duomenų struktūros validumo, medžio šakų semantikos bei kryžminės entropijos sandauga. Duomenų rinkinio dydis: 2500.

   Toliau modelis apmokomas naudojantis IV iteracijos nustatytais svoriais ir sudėtingesniais medžių tipais. Pasirinkti tipai yra skirti kintamųjų deklaravimui kode. Kintamųjų deklaravimas gali būti tiek paprasta, tiek kompleksinė operacija, tačiau *JavaScript* kalboje ji gali būti įgyvendinta be aplinkinių dalykų (pvz. funkcijų). Šių tipų rezultatus tikslinga testuoti pasinaudojant apmokytu modeliu.

7 lentelė. Nuostolio reikšmės per epochą (V iteracija)

Epocha	Nuostolis
1	0,2705
2	0,1733
3	0,0671
4	0,0483
5	0,0508
6	0,0625
7	0,0467
8	0,0301
9	0,0291
10	0,0415

Rezultatai matomi 6 lentelėje, geriausia nuostolio reikšmė pasiekta 9-oje epochoje, tai rodo, kad modelis beveik iki pat galo mokėsi ir tobulėjo. Taip pat, 41 pav. matosi, kad modelis geriau atpažįsta *YAML* struktūrą, kai yra apmokomas ant kelių konkrečių tipų.



41 pav. IV ir V iteracijų *YAML* struktūros validumo rezultatai

Modelį apmokius per kelias iteracijas buvo atrinkti parametrai, kurie bus naudojami apmokant pagrindinį modelį, skirtą generuoti abstrakčius sintaksės medžius. Atlikti testus buvo panaudota 11500 skirtingų medžių iš paruoštų 600 tūkst. medžių.

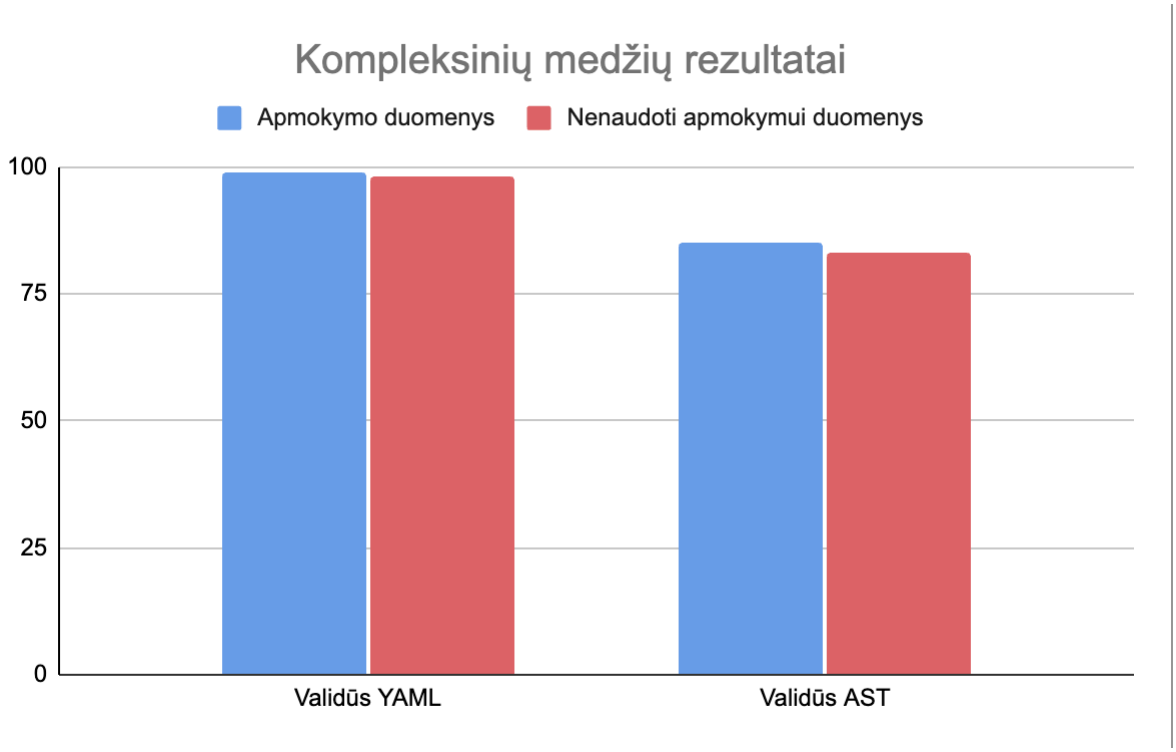
Modelis bus apmokomas naudojantis šiais parametrais: 10 epochų, ankstyvojo stabdymo funkcija su 5 epochų tolerancija, Levenšteino atstumo santykis (3 lentelė), *YAML* duomenų struktūros validumas, medžio semantikos taisyklių tikrinimas 3 šakų gylyje. Modelis bus apmokomas per kelis kartus - pirmiausia primityvūs tipai (*Literal, Identifier*), toliau sudedamieji tipai (*Property, BlockStatement*) ir galų gale pagrinidiniai tipai, kuriuos modelis generuos (*FunctionDeclaration, VariableDeclaration, IfStatement*).

2.4.5. Rezultatai

8 lentelė. Apmokymo žingsniai

Duomenys	Kiekis (tūkst.)	Dydis (m)	Nuostolis
Literal, Identifier	15	m<100	0,0535
Property	10	m<200	0,0327
VariableDeclarator, BlockStatement	10	m<200	0,0865
Įvairūs tipai	10	m<200	0,0220
Įvairūs tipai	10	200<m<500	0,1035
VariableDeclaration, FunctionDeclaration	15	m<750	0,0163

Galutinis modelis buvo apmokytas dalimis, pateiktomis 8 lentelėje. Pirmų keturių apmokymų rezultatai yra pateikiami grafike 42 pav., jame atvaizduoti *YAML* struktūros validumas bei ASM semantikos korektiškumas.



42 pav. *YAML* ir ASM validumo rezultatai

Paskutinių tipų *VariableDeclaration, IfStatement, FunctionDeclaration* galutiniai modelio sugeneruoti kodo pavyzdžiai pateikiami žemiau:

Įvestis: *Create variable: rest client instance for accounts api*. Rezultatas: *const api = "rest"*. Paaiškinimas: Kintamojo sukūrimas konkretaus domeno API. Modelis išlaikė sukūrimo rezultatus, tačiau nesuprato, kad turėjo būti iškviesta *Express* karkaso *route* funkcija.

Įvestis: *Condition: unauthorized error user check*. Rezultatas: *if (authorized)* . Paaiškinimas: Sąlygos sakinys, skirtas patikrinti ar klientas yra prisijungęs į sistemą. Modelis sugeneravo teisingą sintaksę, tačiau buvo tikėtasi logikos sąlygos sakinyje.

Įvestis: *Write an arrow function: code that logs exited when executed*. Rezultatas: *() => log('exit')*. Paaiškinimas: Trumpa funkcija, kuri parašo į terminalą žinutę *exit*.

Nors modelis ir sugeba išlaikyti teisingą struktūrą pagal užklausą, tačiau rezultatai nevisada yra tikslūs. Taip gali būti dėl to, kad buvo naudotas sąlyginai nedideliam duomenų kiekiui buvo sukurtos anotacijos su *OpenAI* modeliu. Tačiau, apmokytas tinklas puikiai geba išlaikyti *YAML* duomenų struktūros validumą ir abstrakčių sintaksės medžių semantiką.

Rezultatai ir išvados

Šiame darbe buvo analizuoti mašininio mokymosi metodai, skirti generuoti *JavaScript* kalbos programinį kodą. Išnagrinėtos kelios dirbtinių neuroninių tinklų architektūros: generatyviniai besivaržantys neuroniniai tinklai, rekurentiniai neuroniniai tinklai bei transformerių neuroniniai tinklai. Darbe analizuoti du pagrindiniai kodo generavimo būdai - sprendžiant kodo generavimo uždavinį kaip natūralios kalbos generavimo uždavinį arba naudojantis abstrakčiais sintaksės medžiais.

Atlikus dirbtinių neuroninių tinklų analizę buvo nuspręsta naudoti transformerių neuroninius tinklus. Transformerių neuroniniai tinklai yra viena iš naujausių dirbtinių tinklų architektūrų, pirmą kartą pristatyta 2017 metais, generatyvinių besivaržančių neuroninių tinklų architektūra pristatyta 2014, o nagrinėta rekurentinių neuroninių tinklų architektūra *LSTM* - 1997. Transfomerių neuroniniai tinklai yra pranašesni natūralios kalbos uždavinių sprendime, turi gerą atminties mechanizmą uždaviniuose, kur kontekstas yra svarbus bei jie turi trumpesnį apmokymo laiką nei *LSTM* ar *GAN*.

Darbe nuspręsta naudoti abstrakčius sintaksė medžius, siekiant išlaikyti semantiškai teisingą kodo struktūrą. Išnagrinėti 57 skirtingi *JavaScript* abstraktūs sintaksės medžiai bei supaprastina jų struktūrą siekiant palengvinti dirbtinio intelekto apmokymą. Dviejų pagrindinių primityvių (*Literal* ir *Identifier*) tipų medžiai buvo išskaidyti į teksto vienetus, iš medžių buvo panaikinta nereikalinga informacija norint kurti individualius kodo fragmentus bei naudota *YAML* duomenų struktūra, kuri yra paprastesnė ir užimanti mažiau atminties nei *JSON*.

Duomenys buvo renkami iš atvirojo kodo platformos *Github*. Buvo surinkta beveik 600 tūkst. medžių iš darbo autoriaus bei dviejų populiarų *YouTube* platformos kūrėjų kodo saugyklų. Duomenų surinkinimui, apdorojimui ir abstrakčių sintaksės medžių generavimui buvo sukurtas specialus komandinės eilutės sąsają turintis įrankis - interneto griaužėjas. Vėliau, įrankis buvo papildytas duomenų failo sukūrimo, rezultatų testavimo, *Open AI* modelio integracijos bei naudojimo sąsajos komandomis. Įrankis turi du būdus klasifikuoti duomenis - naudotis nustatytomis taisyklėmis įvertinant kiekvieną individualią abstraktaus sintaksės medžio eilutę arba naudojantis *OpenAI GPT-3.5-turbo* modelio *API*, su specialia užklausa, generuoti kodo fragmentų aprašymus. Tinkamai parinkus užklausą, *OpenAI* modelius galima paruošti ne tik kodo fragmentų apibūdinimui, bet ir tiksliai nusakyti norimą anotacijos formatą.

Dirbtiniam neuroniniui tinklui apmokyti buvo naudojamas *Google T5* natūralios kalbos uždaviniams spręsti sukurtas modelis. Modelis buvo apmokomas naudojantis *Python* programavimo kalba, *Tensorflow* biblioteka bei *Google Cloud* debesų kompiuterijos aplinka. Šis modelis gali būti lengvai praplečiamas spręsti naujus uždavinius, kadangi naudoja priešdėlių sistemą - kiekvienas priešdėlis leidžia modeliui suprasti, kurią užduotį jis turi atlikti. Darbe modelis buvo papildytas priešdėliu, nurodančiu modeliui, kad uždavinys yra generuoti *JavaScript* kalbos kodą.

Modelio parametrams atrinkti buvo atliekamos penkios mokymosi iteracijos kalibruojant modelį remiantis gautais rezultatais bei nuostolio funkcijos rezultatu. Modeliui apmokyti buvo naudojama Levenšteino atstumo koeficientą skaičiuojanti, *YAML* duomenų struktūros validumą tikrinanti bei medžio semantikos taisykles tikrinanti nuostolių funkcija. Šių dedamųjų rezultatas kartu su kryžminės entropijos sandauga sukurdavo modelio nuostolio reikšmę. Modelyje taip pat naudotas ankstyvojo sustojimo algoritmas, išsaugantis geriausios epochos svorius, taip apsaugodamas mo-

delį nuo persimokymo problemos. Siekiant identifikuoti persimokymo problemą dirbant su mažais duomenų rinkiniais, buvo atlikta k-dalių kryžminė patikra.

Apmokyto modelio rezultatai 98 % tikslumu generavo *YAML* duomenų struktūros rezultatus bei 83 % tikslumu išlaikė abstrakčių sintaksės medžių semantiką. Šie rezultatai yra puikūs, nors modelis ne iki galo išlaiko kontekstą pateikus užklausas prasmingai generuoti kodą. Atlikus analizę ir tyrimus, gautos šios išvados:

- Į modelį įdiegus nuostolių funkciją, kuri yra susijusi su konkrečiu uždaviniu, galima gauti gerus rezultatus - įdiegus *YAML* struktūros validumo funkciją, modelis teisingai generuodavo 98 % rezultatų;
- Modelis paruoštas spręsti įvairias natūralios kalbos problemas gali būti patobulintas ir papildomai apmokytas siekiant spręsti kodo generavimo problemą;
- Modelį naudinga apmokyti palaipsniui, pradedant nuo paprastesnių struktūrų ir mažesnių duomenų rinkinių;
- Abstrakčių sintaksės medžių generavimas leidžia užtikrinti teisingą kodo sintaksę ir supaprastina modelio apmokymą;
- Naudinga pasinaudoti esamais kodo klasifikavimo sprendimais siekiant paruošti prasmingus duomenų rinkinius;
- Transformerių neuroniniai tinklai tinka spręsti kodo generavimo uždavinius, dėl savo savybių išlaikyti struktūrą ir kontekstą.

Literatūra

[ABL+18] U. Alon, S. Brody, O. Levy, and E. Yahav. Generating sequences from structured representations of code. https://arxiv.org/abs/1808.01400, 2018. tikrinta 2021-05-02.

[FGD18] W. Fedus, I. Goodfellow, and A. M. Dai. Maskgan: better text generation via filling in the _. https://arxiv.org/abs/1801.07736, 2018. tikrinta 2021-06-01.

[GPM+14] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A Courville, and Y. Bengio. Generative adversarial networks. https://arxiv.org/abs/1406.2661, 2014. tikrinta 2021-06-01.

[Gra14] A. Graves. Generating sequences with recurrent neural networks. https://arxiv.org/abs/2005.08025, 2014. tikrinta 2021-05-19.

[HS97] S. Hochreiter and J. Schmidhuber. Long short-term memory. https://direct.mit.edu/neco/article/9/8/1735/6109/Long-Short-Term-Memory, 1997. tikrinta 2021-06-13.

[YN17] P. Yin and G. Neubig. A syntactic neural model for general- purpose code generation. https://arxiv.org/abs/2005.08025, 2017. tikrinta 2021-05-19.

[YN18] P. Yin and G. Neubig. A transition-based neural abstract syntax parser for sematic parsing and code generation. https://arxiv.org/abs/1810.02720, 2018. tikrinta 2021-05-02.

[LGC+20] S. Liu, C. Gao, S. Chen, N. Lun Yiu, and Y Liu. Atom: commit message generation based on abstract syntax tree and hybrid ranking. https://arxiv.org/abs/1912.02972, 2020. tikrinta 2021-05-02.

[LKL+18] X. Liu, X. Kong, L. Liu, and K Chiang. Treegan: syntax- aware sequence generation with generative adversarial networks. https://arxiv.org/abs/1808.07582, 2018. tikrinta 2021-06-01.

[LPW+18] Y. Li, Q. Pan, S. Wang, T. Yang, and E. Cambria. A generative model for category text generation. https://www.researchgate.net/publication/324008839_A_Generative_Model_for_Category_Text_Generation, 2018. tikrinta 2020-11-15.

[RM17] M. Rabinovich and D. Klein M Stern. Abstract syntax networks for code generation and semantic parsing. https://arxiv.org/abs/1704.07535, 2017. tikrinta 2021-05-02.

[Roc18] J. Rocca. Understanding generative adversarial networks (gans). https://towardsdatascience.com/understanding-generative-adversarial-networks-gans-cd6e4651a29, 2018. tikrinta 2021-06-02.

[RSR+20] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. https://arxiv.org/abs/1910.10683, 2020. tikrinta 2022-12-29.

[She20] A. Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. https://arxiv.org/abs/1808.03314, 2020. tikrinta 2021-05-24.

[SKS19] N. Sethi, A. Kumar, and R. Swami. Automated web development: theme detection and code generation using mix-nlp. https://www.researchgate.net/publication/335199608_Automated_web_development_theme_detection_and_code_generation_using_Mix-NLP, 2019. tikrinta 2021-05-08.

[SSS20] A. Svyatkovskiy, S. Fu S. Kun Deng, and N. Sundaresan. Intellicode compose: code generation using transformer. https://arxiv.org/abs/2005.08025, 2020. tikrinta 2021-05-15.

[SZX+20] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang. Treegen: a tree-based transformer architecture for code generation. https://arxiv.org/abs/2005.08025, 2020. tikrinta 2021-05-15.

[T P19] A. Santoro T. P. Lillicrap. Backpropagation through time and the brain. https://www.sciencedirect.com/science/article/pii/S0959438818302009, 2019. tikrinta 2021-05-25.

[Tha18] D. Thakur. Lstm and its equations. https://medium.com/@divyanshu132/lstm-and-its-equations-5ee9246d04af, 2018. tikrinta 2021-05-26.

[VSP+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. https://arxiv.org/abs/1706.03762, 2017. tikrinta 2021-06-06.

[WWJ+21] Y. Wang, W. Wang, S. Joty, and S.C.H. Hoi. Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. https://arxiv.org/pdf/2109.00859, 2021. tikrinta 2022-12-29.