VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

MULTIAGENTNÍ SYSTÉM UČÍCÍ SE PREDIKOVAT UŽIVATELSKÉ INTERAKCE V RÁMCI SMART HOME

MULTI-AGENT SYSTEM LEARNING TO PREDICT USER INTERACTIONS WITHIN SMART HOME

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MARTIN ZELENÁK**
AUTHOR

VEDOUCÍ PRÁCE **doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.**
SUPERVISOR

BRNO 2025

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

164617

Ústav: Ústav inteligentních systémů (UITS)
Student: **Zelenák Martin**
Program: Informační technologie
Název: **Multiagentní systém učící se predikovat uživatelské interakce v rámci Smart Home**
Kategorie: Umělá inteligence
Akademický rok: 2024/25

Zadání:

1. Prostudujte problematiku multiagentních systémů a obklopující inteligence.
2. Prostudujte problematiku automatizace budov a Smart Home a existující přístupy, aplikující multiagentní systémy v řízení.
3. Ve spolupráci s vedoucím zvolte vhodný subsystém Smart Home a specifikujte požadavky na jeho inteligentní řízení. Navrhněte řešení s využitím multiagentního paradigmatu a vhodného middleware tak, aby se průběžně na základě senzorických dat, interakcí uživatele se systémem a omezujících podmínek učil predikovat uživatelské interakce.
4. Navrženou řídicí aplikaci implementujte s využitím vybraného multiagentního middleware.
5. Ověřte funkčnost realizované řídicí aplikace v simulovaném prostředí s realistickými modely senzorů a aktuátorů a s vhodně abstrahovaným modelem budovy a chování uživatelů. Vyhodnoťte dosažené výsledky a diskutujte možnosti přechodu k reálnému nasazení.

Literatura:
Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:
První 3 body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Janoušek Vladimír, doc. Ing., Ph.D.**
Vedoucí ústavu: Kočí Radek, Ing., Ph.D.
Datum zadání: 1.11.2024
Termín pro odevzdání: 14.5.2025
Datum schválení: 4.11.2024

Abstrakt

Tato práce se zabývá návrhem a implementací multiagentního systému schopného předvídat uživatelské interakce s chytrými spínači v prostředí chytré domácnosti. Cílem práce je vytvořit systém, který na základě senzorických dat predikuje následující stav zařízení za účelem automatizace. Pro řešení problému byl navržen hierarchický multiagentní systém tvořený hlavním agentem, predikčními agenty pro jednotlivá zařízení a uživatelskými agenty, kteří predikce filtrují a interpretují. K predikci byly využity neuronové sítě typu LSTM a CfC. Funkčnost systému byla ověřena v simulovaném prostředí s abstrahovaným modelem chytré domácnosti a chování uživatele. Nejlepší dosažené výsledky činily 0.92 u metriky F1Score a 0.72 u AUROC. Systém je navržen jako rozšiřitelný a lze ho integrovat do existujících řídicích systémů. Přínosem práce je návrh rozšiřitelné architektury pro adaptivní automatizaci chytré domácnosti s využitím strojového učení.

Abstract

This thesis focuses on the design and implementation of a multi-agent system capable of predicting user interactions with smart switches in a smart home environment. The goal is to develop a system that uses sensor data to predict the next device state for the purpose of automation. To address the problem, a hierarchical multi-agent system was designed, consisting of a central agent, prediction agents for individual devices, and user agents that filter and interpret predictions. LSTM and CfC neural networks were used for the prediction task. The system's functionality was verified in a simulated environment with an abstracted model of the smart home and user behavior. The best achieved results reached an F1Score of 0.92 and an AUROC of 0.72. The system is designed to be extensible and can be integrated into existing control systems. The contribution of this work is a proposal of a modular architecture for adaptive smart home automation using machine learning.

Klíčová slova

Chytrá domácnost, multiagentní systém, agent, predikce uživatelského chování, neuronová síť, LSTM, LTC, CfC, časová řada, hluboké učení, SPADE, PyTorch, simulace, řízení zařízení

Keywords

Smart home, multi-agent system, agent, user behavior prediction, neural network, LSTM, LTC, CfC, time series, deep learning, SPADE, PyTorch, simulation, device control

Citace

ZELENÁK, Martin. *Multiagentní systém učící se predikovat uživatelské interakce v rámci Smart Home.* Brno, 2025. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Vladimír Janoušek, Ph.D.

Multiagentní systém učící se predikovat uživatelské interakce v rámci Smart Home

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Vladimíra Janouška, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Martin Zelenák
13. května 2025

Poděkování

Děkuji vedoucímu práce doc. Ing. Vladimíru Janouškovy, Ph.D. za jeho odborné vedení, trpělivost a velmi cenné rady a konzultace, které mi velmi pomohli s formulací cílů práce a jejich dosažením. Zároveň děkuji své rodině a přátelům za podporu a povzbuzení v průběhu vypracování práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Chytré domácnosti se stávají čím dál více rozšířenými a mnohá chytrá zařízení dokážou komunikovat mezi sebou i s uživatelem. Můžeme tak například ovládat světla chytrým telefonem nebo nastavit, aby se vytápění zapnulo ještě před naším příchodem.

Chytrá domácnost slibuje větší pohodlí, úsporu energie i času. Dnešní systémy často nabízejí automatizaci na základě předem daných pravidel typu „když nastane A, proveď B“. Takové systémy mohou být užitečné, ale nejsou dostatečně flexibilní pro situace, které nejsou předvídatelné. Proto mě zaujala možnost vytvořit systém, který se dokáže učit z chování uživatele a aktivně se přizpůsobovat aktuální situaci a který se učí sám rozhodovat, co udělat, podobně jako to dělá člověk.

Téma chytrých domácností i umělé inteligence je v posledních letech velmi aktuální. V oblasti automatizace domácností se objevuje stále více komerčních řešení, ale jen málo z nich využívá pokročilé metody strojového učení a adaptivní řízení. Vědecký výzkum se zabývá i tím, jak se zařízení mohou učit a předpovídat chování uživatele. Do budoucna se předpokládá, že podobné systémy budou běžnou součástí domácností a pomohou nejen se zvyšováním komfortu, ale i s efektivnějším využíváním energie.

Cílem této práce je navrhnout a implementovat systém pro chytrou domácnost, který dokáže na základě dat z prostředí a uživatelských interakcí predikovat budoucí akce, konkrétně stav chytrých spínačů. Tento systém bude založen na konceptu spolupracujících agentů, z nichž každý má svou roli, a na použití neuronových sítí pro učení a predikci. Navržený systém by měl být zároveň modulární a připravený pro rozšíření nebo nasazení do reálného prostředí.

V kapitole 2 jsou popsána teoretická východiska, která zahrnují koncepty multiagentních systémů a neuronových sítí. Kapitola 3 se věnuje existujícím řešením v oblasti chytrých domácností, které se zabývají predikcí a automatizací. V kapitole 4 je systém navržen a kapitola 5 pak popisuje jeho implementaci jako celku a jednotlivých agentů. Kapitola 6 se zabývá simulací prostředí a uživatelského chování, na jejímž základě se provede ověření učení a funkčnosti systému. Kapitola 7 popisuje použití neuronových sítí a experimenty s nimi. V kapitole 8 jsou uvedeny výsledky experimentů a vyhodnocení systému. Závěr práce v kapitole 9 shrnuje dosažené výsledky a možnosti dalšího rozvoje.

Kapitola 2

Teoretická východiska

V této kapitole jsou popsána nastudovaná teoretická východiska a koncepty. Multiagentní systém se musí umět v čase adaptovat a predikovat chování uživatelů, a k tomu je třeba nějaký učící mechanismus. Způsobů strojového učení je mnoho, ale v této práci se chci zaměřit na využití neuronových sítí.

2.1 Multiagentní systém

Multiagentní systém je distribuovaný výpočetní systém, který se skládá z prostředí a vícero interagujících výpočetních elementů, tzv. agentů. Jako agenta si můžeme představit např. robota nebo virtuálního agenta. Každý agent je autonomní a má schopnost se rozhodovat o svých akcích, aby splnil určitý účel. V rámci multiagentního systému mají agenti schopnost vzájemné komunikace, díky které si mohou vyměňovat nejen data, ale mohou také vést konverzace a domlouvat se za účelem řešení konfliktů a dosažení společných a individuálních cílů. Obsah této kapitoly se opírá o [24, 18, 25].

2.1.1 Prostředí

Jako prostředí můžeme vnímat vše, co agenta obklopuje a co může, alespoň z části, vnímat skrze senzory. U prostředí můžeme zkoumat několik vlastností [24, 25]:

- **Dostupné vs nedostupné** – V dostupné prostředí má agent k dispozici úplný, přesný a aktuální stav prostředí. Většina reálných prostředí jsou z tohoto pohledu nedostupné.
- **Deterministické vs nedeterministické** – V deterministickém prostředí má každá akce právě jeden garantovaný následek. V nedeterministickém prostředí může mít jedna akce různé následky a vzniká tak nejistota o stavu prostředí po provedení akce.
- **Statické vs dynamické** – Statické prostředí nemění svůj stav mimo změn přímo způsobené akcí agenta. Stav dynamického prostředí může být ovlivněn vnějšími vlivy/procesy mimo kontrolu agenta. Fyzický svět je velmi dynamické prostředí.
- **Diskrétní vs spojité** – Množina stavů prostředí je diskrétní/spojitá množina. Číslicový počítač je diskrétní systém a naprogramovaní agenti tak vnímají prostředí diskrétně.

2.1.2 Agent

Agenti mohou pomocí **senzorů** vnímat prostředí, ve kterém se nacházejí, a skrze **aktuátory**, případně **efektory**, s prostředím interagovat. Efektory jsou fyzická zařízení, která přímo manipulují s fyzickým prostředím, např. spínač, a aktuátory zasílají příkazy efektorům. Agenti mají také k dispozici vlastní paměť nazývanou **báze znalostí**.[18]

Jeden z možných přístupů, jak charakterizovat agenta, je podle taxonomie Franklin & Greaser:

- Je **autonomní** – Jeho chování nelze přímo ovlivnit jeho okolím a má plnou kontrolu nad svými akcemi, pokud se jí nevzdá.
- Je **reaktivní** – Adekvátně a pohotově reaguje na změny okolí.
- Je **proaktivní** – Umí projevit iniciativu a konat akce za účelem jeho cílů.
- Má **dočasně spojité chování** – Umí se chovat spojitě, tj. bez dlouhých prodlev.
- Má **sociální schopnosti** – Umí pracovat ve skupině a podílet se na řešení konfliktů.
- Je **adaptivní** – Má schopnost učení a přizpůsobit se.
- Je **mobilní** – Umí se v prostředí pohybovat.
- Je **flexibilní** – Není řízen předepsanými skripty.
- Má **osobnost** – Jeho chování je ovlivněno nejen okolím, ale také jeho osobnostními a emočními rysy.

Důležitou součástí agenta je jeho chování, podle kterého můžeme agenty kategorizovat. Následuje krátký přehled o běžných kategoriích agentů a detailněji jsou popsány v [24, 25].

Reaktivní agenti

Agent ke svým akcím využívá pouze aktuální stav prostředí, jeho vnitřní stav a základní logiku k provedení akce. Neuchovává si tak žádný model/interní reprezentaci prostředí. Jeho akce jsou tak přímým mapováním z pozorovaného stavu prostředí. Tito agenti tak postrádají spoustu z výše uvedených vlastností. [24, 25]

V dynamickém prostředí jsou adaptivní, protože pouze přímo reagují na změny jeho stavu. Ovšem nemají přímo zadaný konkrétní cíl a nemusí tak splnit svůj účel, pro který byli vytvořeni. Dále pak postrádají sociální schopnosti a nemohou tak spolupracovat s ostatními agenty. Chování takového agenta lze implementovat např. konečným automatem. [24, 25]

Racionální agent

Agent využívá svých znalostí a vyvozených domněnek o prostředí pro volbu nejlepší akce, která vede k dosažení jeho zadaného cíle nebo jím stanovených podcílů. Racionální agenti mohou na základě znalostí a vyvozených domněnek předem plánovat své akce. [24, 25]

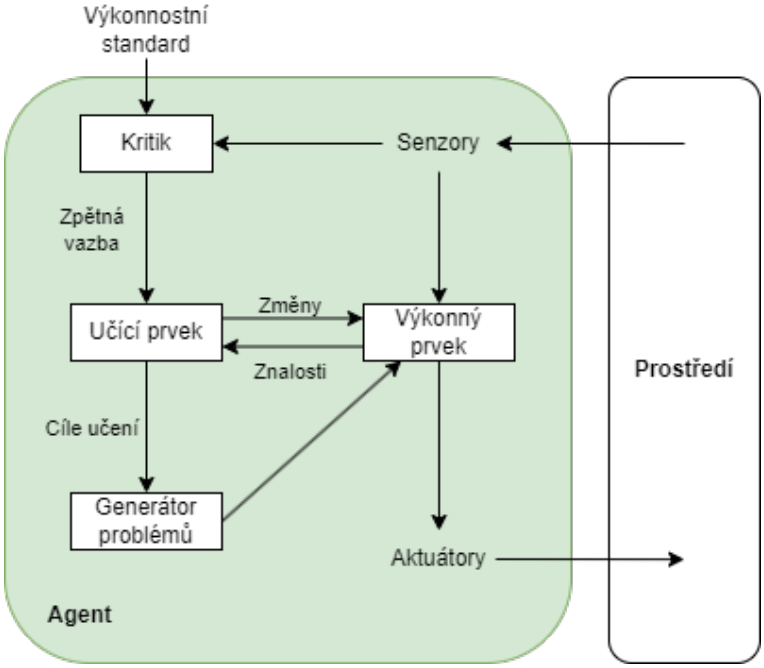
Agent s intencemi

Tito agenti jsou řízeni záměrem. Záměr je pro agenta nějaký přetrvávající cíl, který se snaží dlouhodobě splnit, a to i když není v aktuálním stavu schopen ho dosáhnout. Tito agenti vycházejí z myšlenek Michaela Bratmana, který definuje záměr následovně [25]:

- Záměr je pro agenta problém, pro který musí stanovit, jak ho dosáhnout.
- Záměr vymezuje míru přípustnosti pro další záměry.
- Agent si pamatuje úspěšnost svých pokusů o dosažení záměru.
- Agent nepochybuje o splnitelnosti záměru.
- Agent nepochybuje o jeho schopnosti se přiblížit dosažení záměru.
- Agent, za určitých podmínek, věří, že se přiblížil k dosažení cíle.
- Agent může způsobit nežádoucí vedlejší účinky během dosahování záměru.

Učící se agent

Výhodou učících se agentů oproti výše zmíněným agentům je jejich schopnost se učit a přizpůsobovat se tak i neznámému prostředí. S učením mohou být agenti kompetentnější než pouze se svými počátečními znalostmi. Učící agent lze rozdělit na několik konceptuálních částí znázorněných na obrázku 2.1. [18]



Obrázek 2.1: Obecný učící se agent [18]

Výkonný prvek vybírá akce na základě vjemů z prostředí (senzorických dat). *Učící prvek* je zodpovědný za zlepšování chování. Ze zpětné vazby od *Kritika* určuje, jak by měl agent změnit své chování (úprava *Výkonného prvku*). *Kritik* poskytuje zpětnou vazbu, jak dobře

se agent chová, podle pevně daného *Výkonnostního standardu* a senzorických dat. *Výkonnostní standard* určuje, co je pro agenta dobré a co špatné. *Generátor problémů* navrhuje akce, které vedou k získání nových informací k učení. Agent tak nebude pouze provádět dosud nejlepší naučené akce, ale bude zkoušet i experimentovat, což může vést k lepšímu naučenému chování. [18]

2.1.3 Komunikace

Agenti mohou v rámci multiagentního systému spolupracovat, řešit konflikty anebo argumentovat. K tomuto účelu slouží schopnost komunikace. Pro komunikaci je třeba definovat nějaký jazyk, kterým budou agenti komunikovat. Standardizovaným jazykem je *FIPA Agent Communication Language* [6], který vychází z jazyka *KQML* a definuje, mimo jiné, strukturu a sémantiku zpráv. Komunikace je založena na *řečových aktech* z teorie řečových aktů od L. J. Austina, které jsou v rámci FIPA ACL nazývány *performativní akty* nebo *performativa*. Performativa specifikují komunikační záměr zprávy (typ zprávy), jako např. žádost o provedení akce (request), informační (inform) nebo selhání při provádění akce (failure). FIPA ACL definuje, včetně performativa, několik dalších atributů zpráv. Následuje výčet jen některých důležitých atributů v tabulce 2.1. [25, 6]

Atribut	**Popis**
performative	Typ zprávy
sender	Identifikátor odesílatele
receiver	Identifikátor příjemce
content	Obsah zprávy
language	Jazyk obsahu zprávy
ontology	Označuje kontext/význam obsahu zprávy
reply-with	Výraz, který má příjemce odeslat v odpovědi v atributu in-reply-to
in-reply-to	Výraz, kterým odesílatel odpovídá příjemci

Tabulka 2.1: Důležité atributy zpráv v FIPA ACL

Jediným povinným atributem je *performative*, ovšem dá se očekávat, že většina zpráv bude také obsahovat atributy *sender*, *receiver* a *content*. Atribut *ontology* značí tzv. *ontologii*, která spolu s atributem *language*, značícím jazyk, určují, jak interpretovat obsah zprávy. Jazyk může být např. JSON[1] a ontologie může být např. pozdrav, což by značilo, že obsah zprávy je pozdrav ve formátu JSON. Předpokládá se, že všichni odesílatelé a příjemci znají daný jazyk a ontologii zprávy se stejnou interpretací. Atributy *reply-with* a *in-reply-to* slouží k identifikaci odpovědi na předešlou zprávu. Odesílatel zprávy do atributu reply-with vloží výraz, který příjemce zprávy vloží do atributu in-reply-to ve své odpovědi. Odesílatel pak dokáže spárovat původní zprávu s přijatou odpovědí. Pokud původní zpráva neobsahovala atribut reply-with, pak odpověď by neměla obsahovat atribut in-reply-to. [6]

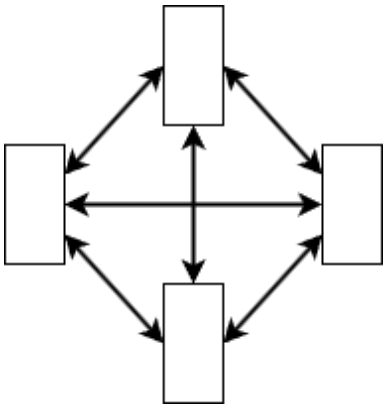
2.1.4 Architektura

Multiagentní systém se může skládat z agentů s různými rolemi, které definují jejich účel v rámci systému. Na základě těchto rolí pak lze definovat, kdo s kým může komunikovat, a tím vytvořit strukturu/architekturu multiagentního systému. V [20] jsou architektury popsány pro agenty využívající *LLM*, lze ovšem tyto architektury využít i obecně:

[1] JSON – https://www.geeksforgeeks.org/what-is-json/

Síť

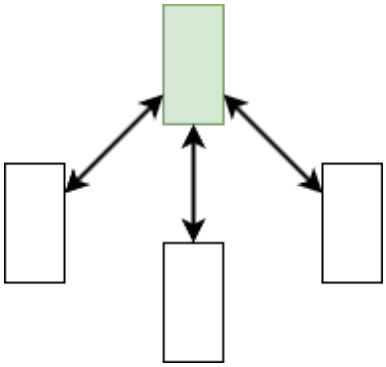
Každý agent může komunikovat s každým. Jedná se o decentralizovanou architekturu, ve které jsou agenti více samostatní a nespoléhají se na centrálního/řídícího agenta pro koordinaci 2.2.



Obrázek 2.2: Síťová architektura

Supervizor

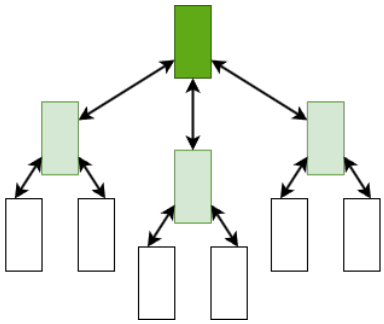
V systému existuje jeden řídící agent, který může komunikovat s každým. Ostatní agenti mohou komunikovat pouze s řídícím agentem. Jedná se o centralizovanou architekturu, kde řídící agent koordinuje a zadává úkoly ostatním agentům 2.3.



Obrázek 2.3: Architektura supervizor

Hierarchie

V systému existují řídící agenti pro podřízené řídící agenty a vzniká tak hierarchie. Jedná se o zobecnění architektury supervizor 2.4.



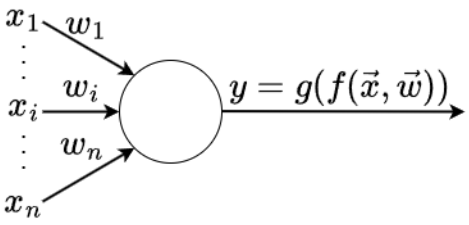
Obrázek 2.4: Hierarchická architektura

2.2 Neuronová síť

Umělé neuronové sítě (dále jen *neuronové sítě*) vycházejí ze základního fungování lidského mozku. Skládají se z *umělých neuronů* (dále jen *neuronů*) a jejich jednosměrných váhovaných vazeb. Strukturou na ně lze obecně nahlížet jako na orientovaný graf.

2.2.1 Neuron

Každý neuron má $x_1, ..., x_n = \vec{x}$ vstupů s váhami $w_1, ..., w_n = \vec{w}$, kde $n >= 1$. Vstupní hodnoty a jejich váhy jsou pak zkombinovány *bázovou funkcí* $f(\vec{x}, \vec{w}) \rightarrow u$, např. *lineární bázová funkce*, která spočítá skalární součin vstupů a vah $u = \vec{x} \cdot \vec{w}$. V literatuře se výstupu bázové funkce též říká vnitřní potenciál neuronu. Výstup bázové funkce je pak vstupem *aktivační funkce* $g(u) \rightarrow y$, jejíž výstup je výsledným výstupem neuronu y, který pak může být vstupem pro další neurony nebo jedním z výstupů neuronové sítě. Umělý neuron je znázorněn na obrázku 2.5. [11]



Obrázek 2.5: Umělý neuron

2.2.2 Topologie

Nejobecnější topologií je *plně propojená síť*, ve které je každý neuron propojen se všemi neurony, a to včetně sebe samého. Váhy mezi dvěma neurony (w_{ij} a w_{ji})[2] mohou být obecně různé. *Symetrická plně propojená síť* je variantou, kde váhy mezi dvěma neurony jsou stejné ($w_{ij} = w_{ji}$). V plně propojené síti mohou být jakékoli neurony vstupní anebo výstupní. Pokud je v síti neuron, který je ovlivněn svým výstupem, a to i cyklem přes více jiných neuronů, označuje se taková síť jako *rekurentní*.

Dalšími topologiemi jsou vrstvené topologie, ve kterých jsou neurony rozděleny do vrstev, které mají určité pořadí od vstupu po výstup neuronové sítě, a kde každý neuron může ovlivňovat pouze neurony ve stejné vrstvě nebo v některé z následujících vrstev. První vrstva sítě se označuje jako *vstupní vrstva* a poslední vrstva se označuje jako *výstupní vrstva*. Ostatní vrstvy, které jsou mezi vstupní a výstupní vrstvou, se označují jako *skryté vrstvy*. Ve vrstvené síti mohou existovat cykly a lze ji tak stále označit jako rekurentní. *Acyklická vrstvená síť* je vrstvená topologie, ve které se nevyskytují vazby mezi neurony ve stejné vrstvě (pouze váhy ${}^{l}w_{ij}$[3], kde neurony i a j jsou v odlišných vrstvách), a proto už není rekurentní. *Dopředná síť* (v angličtině *feedforward network*) je acyklická vrstvená síť, kde existují vazby pouze mezi neurony ze sousedních vrstev (pouze váhy ${}^{l}w_{ij}$, kde neuron j patří do předchozí vrstvy neuronu i). [11, 18]

2.2.3 Učení

Neuronové sítě jsou zajímavé svou schopností se naučit řešit problémy skrze učení/trénování a to i dosud neznámé problémy stejného typu. Tento proces se označuje jako *generalizace*. Naučená/natrénovaná síť se také označuje jako *model*. Teoretických způsobů, jak se síť může učit, je několik, ovšem nejčastějším přístupem je úprava vah mezi neurony. Tímto přístupem lze vazby i rušit nebo znovu vytvářet a to tak, že vazba s váhou 0 nemá na neuron žádný vliv. Neuronová síť je trénována na tzv. *trénovací sadě* dat, kde každý záznam tvoří vstup pro neuronovou síť a případně správný výstup sítě. Trénování pak v krocích (*training step*) prochází trénovací sadou. V každém kroce se nejprve přiloží vstup k vstupní vrstvě neuronové sítě a získá se její výstup, tzv. *dopředná propagace*. Dále se podle typu učení vyhodnotí chyba, které se neuronová síť dopustila [11]:

- *Učení bez učitele* (*unsupervised learning*) – Chyba je spočítána pouze na základě výstupu (řešení) specificky pro daný typ problému. Typicky pro shlukování.
- *Posilované učení* (*reinforcement learning*) – Chyba (nebo odměna) je spočítána na základě stavu prostředí po provedení akce odvozené z výstupu. Typicky pro problémy predikce a rozhodování.
- *Učení s učitelem* (*supervised learning*) – Chyba je spočítána na základě porovnání výstupu s očekávaným správným výstupem, který je součástí trénovací sady.

Následně se vypočítaná chyba propaguje od výstupu ke vstupu, tedy zpětně, skrze neuronovou síť a upravují se váhy jednotlivých vazeb podle toho, jak velký dopad měly na výstup. Tento algoritmus se nazývá *backpropagation*. Na výpočet chyby lze nahlížet jako na funkci, která mapuje váhy vazeb na celkovou chybu sítě, tedy celková chyba pro všechny vstupy. Tuto funkci je pak cílem minimalizovat a to sestupem po jejím gradientu (úpravou vah).

[2] w_{ij} - váha vstupu neuronu i, který je výstupem neuronu j

[3] ${}^{l}w_{ij}$ - váha vstupu neuronu i ve vrstvě l, který je výstupem neuronu j

Minimalizace je pak úkolem optimalizačních metod. Míra, s jakou jsou váhy upravovány, je na základě parametru *míra učení* (*learning rate*), který přímo ovlivňuje rychlost a přesnost učení. [11]

2.2.4 Časové řady

Časová řada je diskrétní posloupnost dat v čase, která může představovat např. teplotu v místnosti. Typicky časové řady tvoří data vzorkovaná s pevným časovým krokem (např. teplota v místnosti každých 5 minut), ale mohou být také tvořeny událostmi, které mezi sebou mají proměnlivý časový krok (např. změna stavu vypínače). Časové řady se dělí podle počtu časově závislých proměnných v datech na *jednorozměrné* (*univariate*) a *vícerozměrné* (*multivariate*). Každý vzorek dat v časové řadě může představovat navzorkovaný stav nějakého systému (jednorozměrný nebo vícerozměrný). Systém pak lze kategorizovat obdobně jako prostředí agenta v podsekci 2.1.1. Jako systém si můžeme představit např. chytrou domácnost a stav systému jako stavy jednotlivých zařízení a senzorická data. Častým problémem reálných systémů je, že nelze získat jeho úplný a přesný stav. [11, 5]

Učení neuronové sítě na základě časové řady se rozděluje na *offline* a *online* učení. Při offline učení je neuronová síť trénována na existující/nashromážděné trénovací datové sadě najednou (*dávkové učení/batch training*) a dále se využívá pouze pro predikci a více se neučí. Po nějaké době, až jsou k dispozici nová data, lze natrénovat nový model nebo upravit stávající model. To je vhodné pro systémy, které jsou deterministické nebo příliš nemění své chování. Při online učení je neuronová síť trénována postupně s nově přicházejícími daty. Síť se tak může postupně adaptovat na změny chování systému, i v případě, že se jeho chování mění. [2, 11]

Úkolem neuronové sítě je predikovat budoucí stav systému, což může být predikce bezprostředně následujícího stavu (*jednokroková predikce*) nebo predikce na více časových kroků dopředu (*vícekroková predikce*). Na vstupu neuronové sítě je n posledních stavů systému, kde $n \geq 1$. Vícekroková predikce lze realizovat dvěma způsoby. Prvním způsobem je *rekurzivní predikce*, kdy neuronová síť udělá několik jednokrokových predikcí, přičemž predikce jednoho kroku se použije jako vstup dalšího kroku. Neuronová síť tak může pokaždé predikovat na jiný požadovaný počet kroků dopředu. Problémem u rekurzivní predikce je kumulovaná chyba a čím více kroků se provede, tím menší je přesnost predikce. Druhým způsobem je *přímá predikce*, kde neuronová síť je během učení trénována, aby predikovala stav přímo na několik kroků dopředu. Nevzniká tak kumulovaná chyba, ale neuronová síť nemůže predikovat stav na libovolný počet kroků dopředu, ale jen na počet kroků, na který byla natrénována. [11]

Neuronovou síť pak můžeme hodnotit podle různých metrik, přičemž záleží, jestli počet možných stavů, které se síť snaží predikovat, je konečný nebo nekonečný. Např. cena cenných papírů na burze má nekonečně mnoho možných hodnot a např. stav spínače má jen dvě možné hodnoty (vypnuto a zapnuto). Predikce některého z konečného množství stavů se označuje jako *klasifikace*. Metriky pro predikci z nekonečně mnoha stavů jsou typicky založeny na vzdálenosti (rozdílu) mezi predikcí a skutečným stavem. Pro klasifikaci ze dvou možných stavů, kde 0 značí negativní hodnotu a 1 kladnou hodnotu, jsou běžné následující metriky:

- *Accuracy* (přesnost) – Poměr správných predikcí ke všem predikcím.
- *Precision* (přesnost) – Poměr správných kladných predikcí ke všem kladným predikcím. Vyjadřuje množství správných kladných predikcí oproti všem kladným predikcím.

- *Recall* (citlivost) – Poměr správných kladných predikcí ke špatným negativním predikcím. Vyjadřuje množství kladných stavů, které síť správně predikovala.
- *F1Score* – Harmonický průměr přesnosti (precision) a citlivosti (recall). Vyjadřuje vyvážení těchto dvou metrik. Vhodné pro nevyvážené množství kladných a negativních stavů. Nabývá hodnot 0.0 až 1.0, kde 0.0 značí přesnost anebo citlivost rovno 0.0 a 1.0 značí pouze správné predikce.
- *AUROC* – Plocha pod ROC křivkou. Vyjadřuje, jak často neuronová síť predikuje skutečně kladné stavy s vyšší hodnotou než predikované hodnoty pro skutečně negativní stavy. Nabývá hodnot 0.0 až 1.0, kde hodnota 0.5 značí náhodné predikce. Hodnota 1.0 značí, že všechny predikce pro skutečně kladné stavy mají vyšší hodnotu než predikce pro skutečně negativní stavy. Lze tedy najít hranici pro predikce, která vede na pouze správné predikce. Hodnoty < 0.5 značí horší než náhodné predikce. Touto metrikou lze zjistit, jak dobře neuronová síť rozlišuje stavy nezávisle na hranici predikce.
- *Hammingova vzdálenost* – Hammingova vzdálenost mezi predikovaným a skutečným stavem. Hammingovi vzdálenosti všech predikcí pak lze zprůměrovat.

Všechny výše zmíněné metriky pak lze rozšířit i pro více než dva stavy a pro vícerozměrné stavy. [23, 15]

2.2.5 Typy neuronových sítí

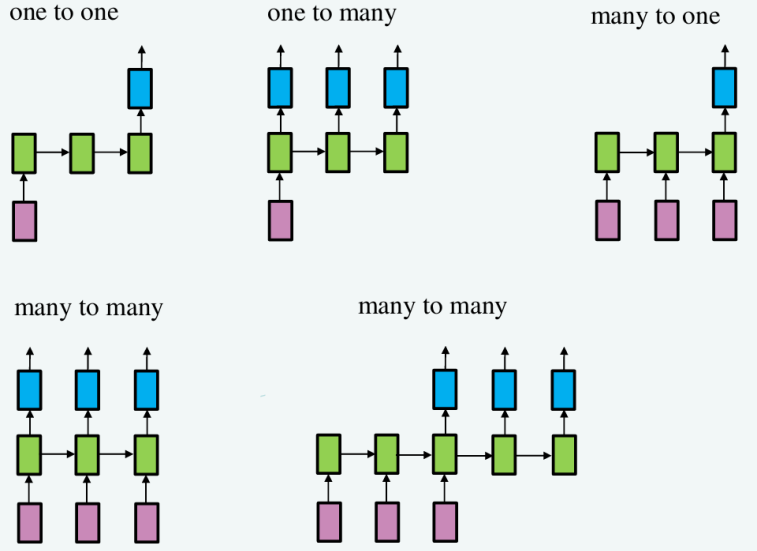
Existuje mnoho různých architektur neuronových sítí, každá vhodná pro jinou sadu problémů. V rámci této práce, jak je naznačeno v podsekci 2.2.4, se zaměřuji na problém predikce uživatelských interakcí v rámci Smart Home jako na problém predikce časové řady. V této podkapitole jsou tak zmíněny jen některé typy neuronových sítí, pro které jsem se v této práci rozhodl.

Plně propojená síť

Plně propojená síť (*fully connected network / FCNN / FC / multilayer perceptron*) je jednou z nejzákladnějších architektur neuronových sítí. Skládá se ze vstupní vrstvy, výstupní vrstvy a skrytých vrstev, kde každý neuron má na vstupu výstupy všech neuronů předchozí vrstvy a jeho výstup je přiveden na vstup každého neuronu následující vrstvy. [11, 19]

Rekurentní síť

Rekurentní síť (*recurrent nerual network / RNN / recurrent multilayer perceptron*) je dopředná vrstvená neuronová síť, kde výpočet dopředné (i zpětné) propagace probíhá v cyklech (časových krocích), kde každý cyklus představuje jeden průchod sítí od vstupu po výstup. Mezi cykly může síť sama sebe ovlivnit výstupy předchozího cyklu. Výstupy předchozího cyklu nemusí být jen přímé výstupy sítě, ale i vnitřní hodnoty, které si síť "pamatuje"do dalšího cyklu. Síť má tedy nějaký vnitřní stav (*hidden state*), skrze který si pamatuje informace napříč cykly. V každém cyklu tak vstup sítě tvoří nejen nový vstup přiložený k její vstupní vrstvě, ale i její vnitřní stav. Zpětná propagace a úprava vah skrze cykly se nazývá *zpětná propagace skrze čas* (*backpropagation through time*), při které dojde k rozložení sítě v čase. Rozložením rekurentní sítě v čase ji můžeme rozdělit do několika kategorií podle délky vstupní a výstupní sekvence znázorněných na obrázku 2.6.

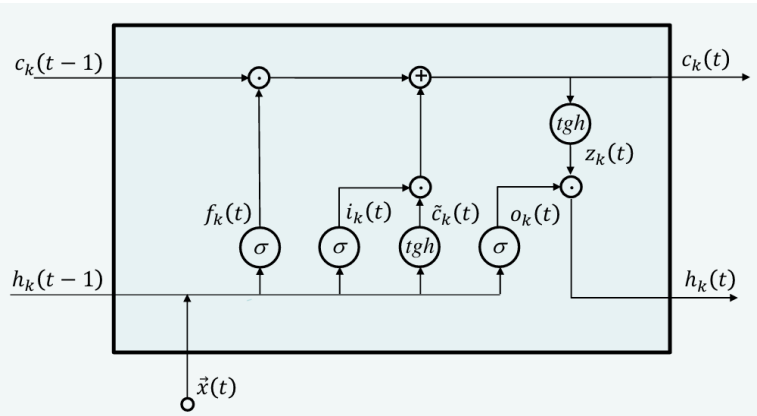


Obrázek 2.6: RNN rozdělení podle délky sekvencí. [26]

Rekurentní sítě jsou využívány, mimo jiné, pro zpracování přirozeného jazyka a predikci časových řad. Tvoří celou třídu neuronových sítí a v této práci budu označovat za rekurentní síť Elmanovu rekurentní síť, kde každá vrstva má na vstupu vstup v aktuálním cyklu a svůj výstup z předchozího cyklu. [11, 19, 26, 17]

LSTM síť

Problémem u zpětné propagace chyby skrz čas u rekurentních sítí je *mizející / explodující gradient*, kde dochází k úbytku až ztrátě nebo nárůstu až "explozi"chyby v čase. To vede k pomalému nebo nestabilnímu učení sítě. Podobný problém pak lze pozorovat i při dopředné propagaci, kde se informace s každým časovým krokem postupně ztrácí. Tyto problémy řeší *dlouhá krátkodobá paměť* (*long short-term memory / LSTM*). LSTM sítě jsou rekurentní sítě, které obsahují *paměťové bloky* s *paměťovými buňkami* (*memory / LSTM cells*) znázorněný na obrázku 2.7. [19, 26]



Obrázek 2.7: LSTM paměťový blok. [26]

Paměťové buňky mají, stejně jako RNN v podsekci 2.2.5, vnitřní stav $h_k(t)$, který se spolu se stavem paměťové buňky předává do dalšího časového kroku. Stav paměťové buňky v čase t je označen $c_k(t)$, který se skládá z předchozího stavu $c_k(t-1)$ a kandidátního stavu $\tilde{c}_k(t)$, kde hodnota kandidátního stavu je v intervalu $< -1, 1 >$ (funkce tgh). Paměťový blok kromě paměťové buňky obsahuje brány, které řídí přístup a pamatování/zapomínání informací v paměťové buňce, jejichž výstupy jsou hodnoty v intervalu $< 0, 1 >$ (aktivační funkce *sigmoid*).

- *zapomínací brána* (*forget gate*) – Určuje kolik informace v paměťové buňce $c_k(t-1)$ se zachová z předchozího kroku.
- *vstupní brána* (*input gate*) – Určuje kolik vstupní informace představované kandidátním stavem $\tilde{c}_k(t)$ se propustí dále / zapamatuje.
- *výstupní brána* (*output gate*) – Určuje jak moc aktualizovaná informace v paměťové buňce $c_k(t)$ ovlivňuje výstup $h_k(t)$.

Výstupem LSTM sítě často bývá přímo vnitřní stav $h_k(t)$, kde t je čas posledního časového kroku. [19, 26]

LTC/CfC síť

Liquid Time-Constant neuronová síť (*LTC*) patří do kategorie tekutých neuronových sítí. Tekutá neuronová síť je rekurentní neuronová síť, která může být plně nebo částečně propojená. Na rozdíl od LSTM, která pracuje s pevnými diskrétními časovými kroky, pracují tekuté neuronové sítě se spojitým časem. Sítě se spojitým časem modelují systém spojitě a ne jen v pevných časových krocích. V LTC má každý neuron, tzv. *tekutý neuron*, vnitřní stav řízený obyčejnou diferenciální rovnicí (*ODR*), která je řešena numerickou metodou, např. kombinací explicitní a implicitní Eulerovy metody. Stav tekutého neuronu je pak určen aktuálním vstupem, minulým stavem a jeho časovým průběhem od minulého do aktuálního časového bodu. Tekuté neurony mají *časově proměnlivé vlastnosti*, které jim umožňují svou reakci na vstup v čase měnit. Tyto vlastnosti jsou dány *tekutými časovými konstantami* (*liquid time-constant*), které určují, jak rychle/pomalu reaguje tekutý neuron na změny ve vstupních datech (signálech). Pokud je časová konstanta krátká, tekutý neuron se rychle přizpůsobuje na nové změny. Pokud je časová konstanta dlouhá, tekutý neuron si déle udrží informaci (pomaleji "zapomíná") a na nové změny se přizpůsobuje pomalu. Tekuté časové konstanty nejsou určeny přímo, ale vycházejí z parametrů, vnitřního stavu a vstupu a mohou se tak v čase dynamicky měnit a tím měnit i reakci neuronu. V průběhu učení pak dochází nejen k úpravám parametrů, ale i k nepřímé úpravě tekutých časových konstant. [8, 10]

V LTC řeší numerické metody ODR v malých integračních krocích, a to může způsobit, že síť je pro reálné nasazení příliš pomalá. *Closed-form continous-time* neuronové sítě (*CfC*) řeší ODR analytickou metodou, která provádí aproximaci výpočtu o jeden až pět řádů rychleji. CfC a LTC sítě jsou vhodné pro náročné problémy časových řad, jako např. autonomní řízení vozidla, predikce lidské činnosti nebo k řízení v oblasti robotiky v dynamickém prostředí. Oproti LSTM dosahují větší přesnosti s menším počtem neuronů a větší rychlosti výpočtu. [7, 10]

Kapitola 3

Existující řešení

Automatizace budov, také označovaná pojmem *chytrá domácnost* (*Smart Home*, *chytré prostředí*), zahrnuje technologie a systémy pro automatizaci a řízení chytrých zařízení v domácnosti. Cílem těchto systémů je zefektivnit komfort, bezpečnost, využití energie a celkovou kvalitu života uživatelů. Mezi typické komponenty chytré domácnosti patří senzory (např. teploměry, detektory pohybu), aktuátory (např. vypínače, termostaty, zámky), centrální řídicí jednotky a software pro vzdálenou správu nebo automatizaci. [3]

Následuje popis některých existujících řešení, která využívají multiagentní přístup anebo strojové učení k automatizaci chytré domácnosti a kterými jsem se alespoň částečně inspiroval. V této práci jsem se rozhodl využít neuronové sítě, a tak jiné způsoby strojového učení v této kapitole nepopisuji.

3.1 MavHome

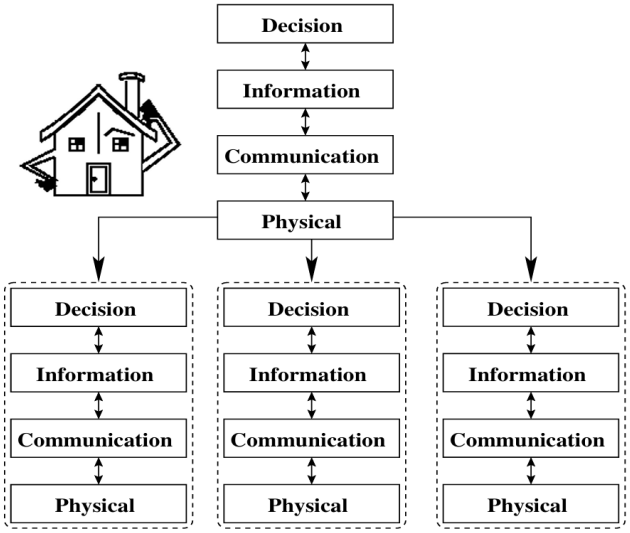
MavHome (*Managing an Adaptive Versatile Home*) je výzkumný projekt, který má za cíl vytvořit architekturu reprezentující chytrou domácnost jako inteligentního agenta. Tento komplexní agent je rozdělen na jednodušší agenty. Každý agent je zodpovědný za daný podúkol v domácnosti a komunikuje s ostatními agenty, aby mohl pozorovat prostředí, rozhodovat se a ovládat zařízení. Architektura rozděluje každého agenta do čtyř vrstev [4]:

- *Rozhodovací vrstva* (*decision layer*) – Vybírá akce, které má agent provést na základě informací z nižších vrstev skrze informační vrstvu.
- *Informační vrstva* (*information layer*) – Shromažďuje, ukládá a generuje informace pro rozhodovací vrstvu.
- *Komunikační vrstva* (*communication layer*) – Zprostředkovává šíření informací, požadavky a celkovou komunikaci mezi agenty.
- *Fyzická vrstva* (*physical layer*) – Obsahuje hardware v chytré domácnosti, např. chytré zařízení, senzory anebo síťový hardware. Protože je celková architektura hierarchická, může fyzická vrstva obsahovat další agenty a tím tak tvořit hierarchii.

Vnímání prostředí probíhá zdola nahoru. Fyzická vrstva prostředí monitoruje skrze senzory a předá informace komunikační vrstvě (např. teplota v místnosti). Komunikační vrstva předá informace informační vrstvě a případně je zašle ostatním agentům. Informační vrstva informace uloží, aktualizuje naučené informace a predikce a informuje rozhodovací vrstvu o nových datech. [4]

Rozhodování a provedení akce pak naopak probíhá zhora dolů. Rozhodovací vrstva vybere akci (např. vypnout topení) a předá rozhodnutí informační vrstvě. Informační vrstva uloží vybranou akci a předá ji komunikační vrstvě. Komunikační vrstva zašle akci konkrétnímu efektoru ve fyzické vrstvě, aby akci vykonal. Efektorem může být i další agent, kterému je informace zaslána jako nová pozorovaná informace. Tento agent pak vybere způsob, jakým akci provést ve své rozhodovací vrstvě a proces se opakuje. Akce může být takto předávána hierarchií až do nejnižší úrovně. [4]

Díky této hierarchické architektuře, znázorněné na diagramu 3.1, mohou agenti fungovat samostatně nebo spolu kooperovat. Zároveň je architektura modulární a MavHome může dynamicky hierarchii přeorganizovat. [4]



Obrázek 3.1: Agentní architektura MavHome. [4]

Jednou z funkcí MavHome je predikce uživatelských akcí na základě pozorování uživatelských aktivit za účelem automatizace repetitivních akcí. K predikci používá kombinaci několika statistických algoritmů [4].

3.2 Silly Home

Silly Home je experimentální rozšíření do řídícího systému Home Assistant[1], které využívá strojové učení k automatizaci. Jeho cílem je predikovat a zautomatizovat uživatelské akce. [9] Každý aktuátor má vlastní model pro predikci, který se periodicky zamění za nově naučený model. Během učení je pro každý aktuátor vytvořeno a trénováno několik nových modelů různých typů [9, 12]:

- Decision Tree Classifier[2]
- Logistic Regression[3]

[1]Home Assistant – https://www.home-assistant.io/
[2]Decision Tree Classifier – https://scikit-learn.org/stable/modules/tree.html
[3]Logistic Regression – https://scikit-learn.org/stable/modules/linear_model.html

- Random Forest Classifier[4]
- Support Vector Machine[5]

Z těchto modelů je pak vybrán ten nejlepší na základě ztrátové funkce a metriky přesnosti (precision). Vstupní vektor (vstup) modelu tvoří aktuální stavy ostatních aktuátorů, senzorická data všech senzorů, denní hodina a den v týdnu. Výstupem modelu je pak predikce nového stavu aktuátoru (vypnout/zapnout). Predikce a provedení akce se spouští s každou změnou stavu prostředí, o které informuje Home Assistant. Silly Home je implementován v jazyce Python a pro strojové učení používá knihovnu scikit-learn (sklearn)[6]. [12]

Získávání senzorických dat, stavů zařízení a jejich ovládání je řešeno skrze Home Assistant. Výhodou využití existujícího řídícího systému je, že Silly Home se nemusí zabývat sběrem dat a komunikací s konkrétními senzory a zařízeními. Získaná data si Silly Home ukládá do své lokální databáze, aby je později mohl využít k natrénování nových modelů. [9]

3.3 Shrnutí

Obě řešení se zaměřují na predikci uživatelských akcí v chytré domácnosti, ale každé jiným způsobem. MavHome využívá hierarchický multiagentní systém, který kombinuje různé statistické metody k predikci sekvencí uživatelských akcí. Silly Home nevyužívá multiagentní systém, ale predikci rozděluje na jednotlivé predikční modely pro každý aktuátor zvlášť, přičemž k predikci využívá klasické metody strojového učení, jako jsou rozhodovací stromy nebo logistická regrese.

Výhodou multiagentní architektury MavHome je její rozšiřitelnost. Stejně tak Silly Home, který predikci rozděluje mezi jednotlivé modely, lze rozšiřovat o nová zařízení. Zatímco MavHome řeší veškeré řízení smart home a je tak řídícím systémem, Silly Home neřeší přímou komunikaci, pozorování a řízení zařízení a spoléhá se na existující řídící systém a funguje jako jeho rozšíření. Šlo by ho tak integrovat i do jiných existujících řídících systémů bez nutnosti vyměnit celý řídící systém.

Cílem práce bude vytvořit multiagentní systém s rozšiřitelnou architekturou inspirovanou MavHome. Tedy systém by měl být za běhu rozšiřitelný o nová zařízení. Zároveň, podle Silly Home, by predikce měla být rozdělena pro každé zařízení zvlášť a systém by měl být integrovatelný do existujícího řídícího systému.

[4]Random Forest – https://scikit-learn.org/stable/modules/ensemble.html
[5]Support Vector Machine – https://scikit-learn.org/stable/modules/svm.html
[6]scikit-learn – https://scikit-learn.org/stable/index.html

Kapitola 4

Návrh systému

Cílem této kapitoly je navrhnout multiagentní systém tak, aby se průběžně učil predikovat uživatelské akce a s předstihem je provedl za účelem automatizace. Návrh je rozdělen na tři samostatné části. První část tvoří návrh samotného multiagentního systému, tedy jak bude systém fungovat jako celek, jaké agenty bude obsahovat a jak spolu budou komunikovat. Druhá část se zabývá experimenty s neuronovými sítěmi a způsobem jejich vyhodnocení a srovnání. Tyto experimenty jsou prováděny s předem vygenerovanými daty ze simulace. Třetí část se tedy zabývá simulací, konkrétně modelem uživatele a jeho interakcemi s prostředím. Spolu s vedoucím jsme zvolili podsystém chytrých spínačů, jako např. chytré osvětlení. Systém by tak měl provádět akce typu vypnout/zapnout. Z hlediska technologií jsem se rozhodl zvolit programovací jazyk *Python*.

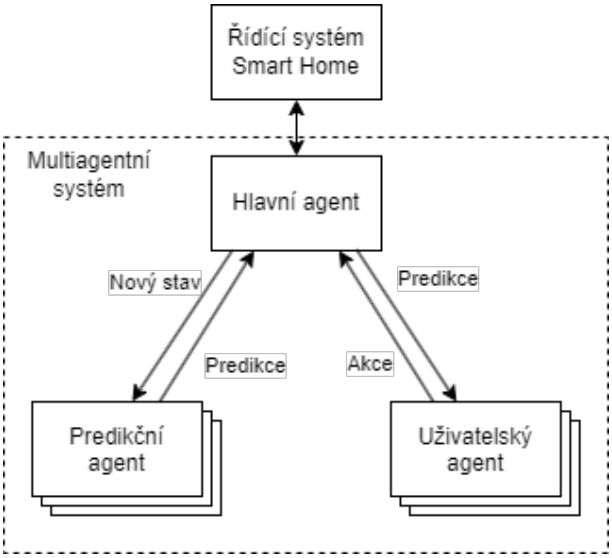
4.1 Multiagentní systém

Návrh multiagentního systému spočívá v určení typů a cílů jednotlivých agentů, architektury, vzájemné komunikace a celkové dynamiky systému. Dále specifikovat vnější rozhraní se systémem a technologie, kterými bude systém implementován.

Multiagentní systém jsem navrhl tak, aby periodicky predikoval stavy spínačů a aby tyto predikce byly filtrovány a interpretovány podle uživatelského nastavení. Každý uživatel může pro každé zařízení nastavit vlastní filtraci a interpretaci. Systém je zároveň za běhu rozšiřitelný o nové zařízení a uživatele. Interakce s multiagentním systémem je realizována skrze vnější rozhraní, které může být integrováno přímo do existujícího řídícího systému Smart Home (*vnější řídící systém*/*řídící systém*). Základní funkcionalita rozhraní je spouštění a vypnutí multiagentního systému a zapouzdření komunikace. Detailnější popis rozhraní je spíše implementační detail, a tak zde není více popsáno a uvažuje se, že zprávy zasílá přímo vnější řídící systém.

4.1.1 Architektura

V této podkapitole je popsána architektura multiagentního systému, tedy jaké agenty obsahuje, jaký je jejich účel a jak spolu interagují. Tyto agenty lze považovat za spíše reaktivní agenty, jelikož jen přímo reagují na zprávy na základě svého vnitřního stavu anebo báze znalostí. Je třeba podotknout, že predikční agent mění svůj vnitřní stav trénováním neuronové sítě a uživatelský agent může změnit svou vnitřní logiku při změně uživatelského nastavení. Tím by se tito agenti mohli vymykat reaktivním agentům. Architektura je znázorněna na diagramu 4.1.



Obrázek 4.1: Architektura multiagentního systému

Hlavní agent

Hlavní agent koordinuje a řídí multiagentní systém a centralizuje komunikaci, tedy každý agent bude pouze komunikovat s centrálním agentem. Architektura systému je tedy *architektura supervizor*, kde hlavní agent je supervizorem. Dalším účelem hlavního agenta je komunikace s vnějším systémem, ve kterém bude multiagentní systém integrován. Hlavní agent primárně rozesílá nové stavy prostředí predikčním agentům, jejich predikce zasílá uživatelským agentům a jejich akce do vnějšího systému k provedení. Vnějším systémem může být např. rozšíření do řídicího systému *Home Assistant*, tak, jak je to v případě *Silly Home* v sekci 3.2. Tento způsob centralizované komunikace zjednodušuje její návrh a implementaci výměnou za decentralizovanost multiagentního systému. Další úlohou hlavního agenta je vytváření nového predikčního agenta a uživatelského agenta, když je oznámeno nově přidané zařízení nebo uživatel vnějším systémem. Multiagentní systém tak bude schopný se za běhu rozšiřovat.

Predikční agent

Mojí prvotní myšlenkou bylo navrhnout jednoho predikčního agenta, který by obsahoval model, jenž by přímo predikoval dvojici zařízení, akce (klasifikace zařízení a klasifikace akce). Ve srovnání s existujícími řešeními by ovšem tento přístup nevyužíval sílu multiagentních systémů. Zároveň už první prototyp ukázal, že jde o příliš složitý problém pro jeden model (alespoň ne rozumně velký model) a predikoval by akci pro pouze jedno zařízení pro aktuální vstup. Lepším přístupem je vytvořit predikčního agenta pro každé zařízení zvlášť, který predikuje následující stav daného zařízení (vypnuto/zapnuto). Tento přístup už využívá několika agentů pro rozdělení původně složitého problému na jednodušší části a podobá se přístupům existujících řešení.

Predikční agent periodicky generuje nové predikce stavu zařízení na základě stavu prostředí, které mu zasílá hlavní agent. Vygenerované predikce jsou výstupní hodnoty modelu a nejsou nějak interpretovány a jsou zasílány hlavnímu agentovi. Predikční agent také průběžně svůj model učí. Tyto agenty také někdy označuji jako *agenty zařízení*.

Uživatelský agent

Posledním typem agenta je *uživatelský agent*, který v systému reprezentuje konkrétního uživatele. Úlohou tohoto agenta je filtrovat a interpretovat predikce od predikčních agentů z pohledu daného uživatele a generovat akce. Filtrace a interpretace jsou na základě uživatelského nastavení. Uživatel může pro každé zařízení nastavit:

- Povolení – Jestli je pro dané zařízení povoleno, aby ho multiagentní systém spínal.
- Hraniční hodnota vypnutí – Pokud je hodnota predikce menší než hraniční hodnota vypnutí, je predikce interpretována jako vypnutí a je vygenerována akce.
- Hraniční hodnota zapnutí – Pokud je hodnota predikce větší než hraniční hodnota zapnutí, je predikce interpretována jako zapnutí a je vygenerována akce.

Vygenerovaná akce je dále zaslána hlavnímu agentovi, aby ji zaslal do řídícího systému k provedení. V případě, že zařízení není nastaveno, spínání není povoleno nebo hodnota predikce je mezi nastavenými hraničními hodnotami, není vygenerována žádná akce.

4.1.2 Zprávy

V této podkapitole jsou popsány zprávy, které agenti zasílají nebo přijímají. Zprávy jsou navrženy tak, aby využívaly standardu FIPA ACL [6]. Všechny zprávy mají podle FIPA ACL povinný atribut *performative* a atributy *content* (obsah), *sender* (odesílatel), *receiver* (příjemce) a *ontology* (ontologie), případně mohou obsahovat atribut *reply-with* a vyžadovat tak odpověď s atributem *in-reply-to*. Atribut performative určuje typ zprávy, tak jak specifikuje FIPA ACL, jako např. *inform* (informační) nebo *request* (požadavek). Atribut ontology určuje, o jakou konkrétní zprávu jde, jako např. *newState* nebo *prediction*. Některé zprávy mají navíc atribut *language*, který určuje formát obsahu zprávy, jako např. *JSON*.

Nový stav prostředí

Zpráva o novém stavu prostředí je zasílána vnějším řídícím systémem hlavnímu agentovi a obsahuje nový stav prostředí. Řídící systém by tuto zprávu měl zaslat vždy, když se stav prostředí změní. Hlavní agent pak tuto zprávu přepošle všem predikčním agentům. Navržené atributy zprávy jsou v tabulce 4.1.

Atribut	**Hodnota**
performative	*inform*
content	Nový stav prostředí
language	Formát nového stavu
ontology	*newState*

Tabulka 4.1: Atributy zprávy o novém stavu prostředí

Predikce

Zpráva s vygenerovanou predikcí je zaslána predikčními agenty hlavnímu agentovi a obsahuje výstupní hodnotu predikčního modelu bez jakékoli interpretace. Hlavní agent tuto zprávu přepošle všem uživatelským agentům. Protože se systém zaměřuje na chytré spínače a stav zařízení je tak jedna hodnota (vypnuto/zapnuto), bude obsahem zprávy pouze jedno

číslo a identifikátor zařízení. Není tak nutné specifikovat formát obsahu zprávy (atribut language). Navržené atributy zprávy jsou v tabulce 4.2.

Atribut	**Hodnota**
performative	*inform*
content	Identifikátor zařízení a hodnota predikce
language	–
ontology	*prediction*

Tabulka 4.2: Atributy zprávy s predikcí

Vykonání akce

Zpráva s požadavkem o vykonání akce je zaslána uživatelskými agenty hlavnímu agentovi a obsahuje trojici: časové razítko, identifikátor zařízení a akci. Časové razítko určuje, kdy byla akce vygenerována / kdy se má vykonat. Akce je pak v případě chytrých spínačů, tak jako u predikce, jen jedna hodnota, která už je interpretována jako konkrétní stav (1 = zapnout, 0 = vypnout). Hlavní agent pak tuto zprávu přepošle vnějšímu řídícímu systému k vykonání. Navržené atributy zprávy jsou v tabulce 4.3.

Atribut	**Hodnota**
performative	*request*
content	Časové razítko, identifikátor zařízení a akce
language	–
ontology	*takeAction*

Tabulka 4.3: Atributy zprávy s predikcí

Nastavení uživatelského agenta

Zpráva s uživatelským nastavením pro konkrétní zařízení je zaslána vnějším řídícím systémem uživatelskému agentovi. Tato zpráva je určena uživatelskému agentovi a je mu zaslána přímo. Výjimečně tak zpráva není zaslána hlavnímu agentovi. Zpráva obsahuje identifikátor zařízení a uživatelské nastavení pro filtrování a interpretaci predikcí pro toto zařízení. Navržené atributy zprávy jsou v tabulce 4.4.

Atribut	**Hodnota**
performative	*request*
content	Identifikátor zařízení, nastavení pro predikce
language	Formát nastavení
ontology	*setDeviceFilter*

Tabulka 4.4: Atributy zprávy s uživatelským nastavením pro zařízení

Přidání nového predikčního agenta

Zpráva s požadavkem o přidání nového predikčního agenta pro nové zařízení je zaslána vnějším řídícím systémem hlavnímu agentovi. Hlavní agent pak vytvoří nového predikčního agenta s identifikátorem zařízení obsaženým ve zprávě. Navržené atributy zprávy jsou v tabulce 4.5.

Atribut	**Hodnota**
performative	*request*
content	Identifikátor zařízení
language	–
ontology	*newDevice*

Tabulka 4.5: Atributy zprávy pro přidání predikčního agenta

Přidaní nového uživatelského agenta

Zpráva s požadavkem o přidání nového uživatelského agenta pro nového uživatele. Tato zpráva má obdobný význam jako zpráva s požadavkem o přidání predikčního agenta. Hlavní agent podle identifikátoru uživatele obsaženého ve zprávě vytvoří nového uživatelského agenta. Navržené atributy zprávy jsou v tabulce 4.6.

Atribut	**Hodnota**
performative	*request*
content	Identifikátor uživatele
language	–
ontology	*newUser*

Tabulka 4.6: Atributy zprávy pro přidání uživatelského agenta

Zastavení agenta

Tuto zprávu musí umět přijímat každý agent v systému. Tato zpráva je požadavkem na ukončení agenta. Pokud vnější řídící systém zašle tuto zprávu hlavnímu agentovi, pak hlavní agent musí touto zprávou ukončit všechny agenty a nakonec ukončit sám sebe a tím ukončit celý multiagentní systém. Navržené atributy zprávy jsou v tabulce 4.7.

Atribut	**Hodnota**
performative	*request*
content	*stop*
language	–
ontology	*stop*

Tabulka 4.7: Atributy zprávy pro přidání uživatelského agenta

4.1.3 Technologie

Pro jazyk Python existuje několik frameworků pro realizaci multiagentního systému, ovšem většina z nich se zaměřuje na agenty, kteří obsahují nebo se dotazují nějakého *LLM*[1], což není způsob, jakým navržený systém funguje. Pro tuto práci jsem zvolil SPADE.

SPADE (*Smart Python Agent Development Environment*) je platforma (multiagentní middleware) pro tvorbu multiagentních systémů založená na komunikačním protokolu *XMPP*[2], který poskytuje komunikaci mezi agenty strukturovaným způsobem. Agenti také nemusí být omezeni na pouhý jeden proces či počítač a mohou být distribuováni napříč několika výpočetními uzly a komunikovat mezi sebou. XMPP je zároveň protokol, který je rozšířený mezi chytrými Smart Home zařízeními, a uživatel by tak mohl využít jeho stávající infrastrukturu, konkrétně tedy XMPP server. Komunikační zprávy v SPADE vycházejí z FIPA ACL a přímo tak podporují navržené zprávy. [16]

4.2 Učení

V této sekci jsou navržena vstupní data, použité neuronové sítě a způsob porovnání výsledků experimentů s neuronovými sítěmi pro predikci stavů zařízení ve Smart Home. Cílem učení v systému je predikovat následující stavy zařízení na základě aktuálních a historických dat s rozumným předstihem a tím predikovat uživatelské interakce.

Rozhodl jsem se, že pro učení systému použiji neuronové sítě, protože mi jejich využití přijde zajímavé a myslím, že by stálo za prozkoumání jejich aplikace v oblasti automatizace chytré domácnosti ve spojení s multiagentním systémem.

4.2.1 Vstupní data

Vstupními daty tvoří časovou řadu s pevným časovým krokem. Každý záznam je vstupní vektor (uspořádaná n-tice) pro neuronovou síť reprezentující stav chytrého prostředí. Součástí stavu prostředí jsou stavy jednotlivých zařízení, čas a lokace uživatelů.

Čas

Čas je pro model klíčovou kontextovou informací, která udává, kdy k uživatelské akci došlo. To je důležité z hlediska učení vzorů uživatelských interakcí. Čas je ve vstupních datech reprezentován několika hodnotami:

- Minuta – Počet minut od celé hodiny (0–59)
- Hodina – Počet hodin od počátku dne (0–23)
- Den v týdnu – Číslo dne od počátku týdne (1–7)
- Den v měsíci – Číslo dne od počátku měsíce (1–31)
- Měsíc – Číslo měsíce od počátku roku (1–12)
- Rok

[1] LLM (Large Language Model) – https://www.geeksforgeeks.org/large-language-model-llm/
[2] XMPP – https://xmpp.org/

Tento způsob reprezentace času v několika hodnotách dovoluje modelu se naučit opakující se vzorce uživatelských interakcí s různými časovými měřítky, jako např. denní a týdenní rutiny a rozdílné vzorce v různých obdobích.

Lokace uživatelů

Informace o lokacích uživatelů umožňuje modelu se naučit souvislost mezi uživatelskými interakcemi a místností, ve které se uživatel nachází. Vstup obsahuje lokaci pro každého uživatele, která je reprezentována číslem místnosti. Ačkoli se jedná o užitečnou informaci, může být problém s jejím získáním. Způsoby, jak detekovat přítomnost a lokaci uživatele, jsou popsány v [1]. Data o lokacích uživatelů nemusí být v chytrém prostředí dostupná, a tak by tato informace měla být nepovinná.

Stavy zařízení

Stav zařízení může být složen z více dílčích hodnot, tato práce se ale zaměřuje na chytré spínače, tedy binární stav vypnuto/zapnuto. Lze ho tak reprezentovat jednou hodnotou 0/1. Vstup pak obsahuje jednu hodnotu pro každé zařízení.

Omezení neuronových sítí

Jednou z nevýhod neuronových sítí je, že pro již vytvořenou síť nelze upravit její vstupní vektor aniž by se musela znovu učit. Aby mohl být systém rozšiřitelný o nová zařízení, bude mít každý model na vstupu čas, lokace uživatelů a stav konkrétního zařízení, jehož stav predikuje. Modely se tak nebudou moct naučit souvislosti mezi stavy a změnami stavů všech zařízení.

To samé samozřejmě platí o lokacích nových uživatelů. Systém by však stále měl být rozšiřitelný o nové uživatele, resp. uživatelské agenty. Proto jsem se rozhodl, že vstup bude obsahovat pouze lokace uživatelů, kteří byli do systému přidáni před prvním přidaným zařízením. Později tak lze přidat nové uživatele, resp. jejich agenty, ale jejich lokace nebude při predikci použita.

4.2.2 Neuronové sítě

Pro experimenty a následné použití v multiagentním systému jsem vybral následující neuronové sítě. Tyto sítě jsem vybral na základě vstupních dat, které tvoří časovou řadu, a na problém predikce interakcí/akcí/stavů zařízení tak nahlížím jako na problém predikce časové řady.

Plně propojená síť (FCNN)

Plně propojená síť je jednoduchá dopředná síť pro základní experimenty. Tyto základní experimenty slouží k porovnání s ostatními sítěmi, které by měly dosahovat lepších výsledků. Zároveň by měly ověřit, že simulace generuje dostatečně komplexní data s časovými závislostmi, a to tak, že by jednoduchá plně propojená síť neměla dosáhnout dobrých výsledků.

Rekurentní síť (RNN)

RNN (Elmanova rekurentní síť) oproti FCNN umí zpracovávat sekvence dat a měla by se tak naučit alespoň krátké časové závislosti. Problémem u této jednoduché rekurentní sítě je

problém mizejícího gradientu u dlouhých časových závislostí. Experimenty s RNN, podobně jako s FCNN, slouží k porovnání s ostatními sítěmi a k ověření komplexity generovaných dat.

LSTM síť

LSTM je standardní sítí, která se pro predikci časových řad používá. Oproti RNN netrpí problémem mizejícího gradientu a měla by být schopna se naučit i dlouhodobé časové závislosti v datech, což je vhodné pro predikci vzorců s velkým časovým měřítkem.

CfC síť

Stejné nebo lepší výsledky než LSTM s mnohem menším počtem neuronů. CfC síť je vhodná pro náročné problémy časových řad a měla by dosahovat lepších výsledků než LSTM s menší výpočetní náročností. Její velkou výhodou je, že umí pracovat i s nepravidelně vzorkovanými daty, tedy pracovat s různou délkou časového kroku. CfC by tak měla být síť, která by se při reálném nasazení v systému používala.

4.2.3 Vyhodnocení

Pro vyhodnocení experimentů s jednotlivými neuronovými sítěmi a jejich srovnání jsou v průběhu experimentů zaznamenávány průběhy vstupních hodnot, predikcí a metrik. Metriky jsou zaznamenávány nejen pro celkový predikovaný stav, ale i pro stavy jednotlivých zařízení. Zaznamenávají se následující metriky:

- Accuracy – Procentuální úspěšnost predikce.
- Precision – Množství správně predikovaných zapnutí zařízení ze všech predikovaných zapnutí.
- Recall – Množství správných predikcí ve chvíli kdy je zařízení zapnuté.
- F1Score – Vyvážení mezi Precision a Recall.
- AUROC – Rozlišení stavů vypnutí a zapnutí nezávisle na nastavené hranici predikce.
- Hammingova vzdálenost – Poměr nesprávně predikovaných stavů.
- Přesnost (accuracy) při změně stavu – Procentuální úspěšnost predikce změn stavu zařízení.

4.2.4 Technologie

Pro implementaci experimentů a modelů v multiagentním systému jsem se rozhodoval mezi *PyTorch* [17] a *TensorFlow* [22]. Nakonec jsem zvolil PyTorch, protože je flexibilnější a jednodušší na použití pro experimentování a prototypování, zatímco TensorFlow se spíše hodí pro produkční nasazení.

Pro experimentování jsem dále zvolil *PyTorch Lightning* [14]. PyTorch Lightning je nadstavba pro PyTorch, která strukturuje experimenty a odděluje logiku konkrétního modelu od logiky trénování. Zároveň umožňuje jednoduché zaznamenávání. Jeden z možných výstupů pro záznamy je zaznamenávání do formátu pro *TensorBoard*.

TensorBoard [22] je nástroj pro vizualizaci průběhu experimentů, sledování metrik a srovnání výstupů jednotlivých experimentů. TensorBoard je součástí TensorFlow, ale je kompatibilní i s PyTorch skrze PyTorch Lightning.

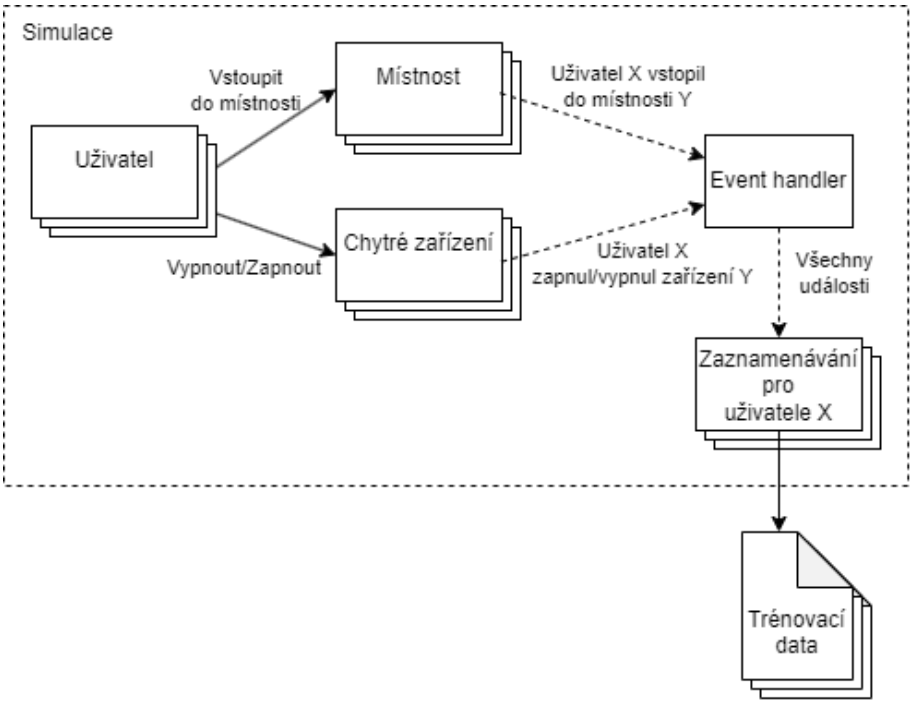
PyTorch obsahuje běžně používané neuronové sítě, ovšem neobsahuje tekuté neuronové sítě, konkrétně CfC síť. Pro implementaci tekutých neuronových sítí využívám GitHub repozitář pro *NCPs* (*Nerual Circuit Policies*) [13], který obsahuje jak PyTorch, tak TensorFlow implementace tekutých neuronových sítí. Zároveň obsahuje mechanismy pro vytváření vazeb mezi tekutými neurony.

4.3 Simulace

Před vytvořením multiagentního systému a jeho učením je třeba mít k dispozici data, na kterých lze systém učit a později ověřit jeho funkcionalitu. V reálné chytré domácnosti by však trvalo příliš dlouho získat potřebné množství dat. Za tímto účelem je tak vhodné vytvořit simulované prostředí, které je nejen rychlejší než reálný svět, ale také lze parametrizovat.

4.3.1 Architektura

V této podkapitole je popsána architektura simulace, tedy jak je modelováno Smart Home prostředí, chytrá zařízení a uživatelé a jak spolu vzájemně interagují. Simulace je navržena jako diskrétní simulace. Architektura se skládá z několika navzájem propojených komponent a je znázorněna na diagramu 4.2.



Obrázek 4.2: Architektura simulace

Uživatel je model konkrétního uživatele, který se ve Smart Home pohybuje a interaguje s ním, a modeluje tak uživatelské chování a interakce. *Místnost* je model lokace, kde se uživatel může nacházet (např. kuchyň, ložnice nebo mimo domov), a obsahuje chytrá zařízení.

Model uživatele pak může do místnosti vstoupit nebo skrze ni přistoupit k zařízením, které obsahuje. *Chytré zařízení* je model chytrého zařízení, v této práci chytrý spínač, který obsahuje vnitřní stav a definuje operace, které lze se zařízením provádět (vypnout/zapnout). Modely místností a chytrých zařízení při uživatelské interakci generují události. Každá událost je předána komponentě *Event handler*, která pak událost přepošle komponentám, které jí naslouchají. Událostem naslouchají komponenty, které provádějí *zaznamenávání pro uživatele X*, kde *X* je nějaký konkrétní uživatel. Tyto komponenty periodicky logují informace odpovídající vstupním datům neuronových sítí do výstupního souboru, na základě kterého mohou být prováděny experimenty s neuronovými sítěmi.

4.3.2 Model uživatele

Nejdůležitější součástí simulace jsou modely uživatelů, resp. modelování jejich chování. Uživatelské chování musí být, tak jako reálné chování uživatele, ovlivněné/omezené časem. Zároveň by se uživatel měl chovat každý den jinak podle různých okolností. V této podkapitole nejprve popíši, jak chování uživatelů modeluji, a následně dva konkrétní modely.

Uživatelské chování jsem se rozhodl rozdělit na dvě chování: v pracovní den a o víkendu. Dny se pak skládají z aktivit, ve kterých může uživatel interagovat s chytrými zařízeními. Chování modeluji pomocí různých stavů reprezentujících aktivity uživatele, mezi kterými model přechází. V každém stavu je možné přejít do jednoho z více stavů na základě pravděpodobnosti, kde součet pravděpodobností přechodů je 100%. Dále má každý stav dvě časová omezení:

- Časově omezený vstup – Časový interval, kdy je možné do stavu vstoupit, např. od 13:00 do 15:00.
- Setrvání ve stavu – Minimální a maximální čas dne, do kdy je nutné/možné ve stavu zůstat, např. minimálně do 13:00 a maximálně do 15:00.

V základu se tak jedná o konečný automat, který jsem ale rozšířil o přechody podmíněné pravděpodobností a časová omezení. Dále má každý stav svou stochastickou dobu trvání podle aktivity, kterou modeluje. Pokud by stav během simulace trval příliš krátkou dobu a nesplnil tak omezení setrvání ve stavu, musí model počkat do minimálního času. Naopak pokud by stav měl překročit maximální čas, musí být zkrácen.

Stochastický model

Tento model by měl připomínat možné reálné chování uživatele ve Smart Home a bude sloužit k ověření funkcionality multiagentního systému. Téměř v každém stavu uživatel interaguje s chytrými zařízeními a setrvá v něm po stochasticky dlouhou dobu. Model obsahuje stavy vypsané v tabulce 4.8. Chování v pracovní den a o víkendu jsou znázorněny na diagramech 4.3 a 4.4, přičemž *t* je čas dne (hodiny:minuty), *Od:* značí časově omezený vstup a *Do:* značí omezení na setrvání ve stavu.

Stav	Popis
SLEEPS	Spí
WAKES UP	Vzbudí se
PREPARES TO LEAVE	Chystá se odejít s budovy
LEFT	Odešel z domu
ARRIVES	Přišel do domu
RELAXES	Relaxuje
READS	Čte si
WORKS	Pracuje v domě
DOES HOBBY	Dělá svůj koníček
PREAPARES FOOD	Připravuje si jídlo
EATS	Jí připravené jídlo
GOES TO SLEEP	Chystá se spát

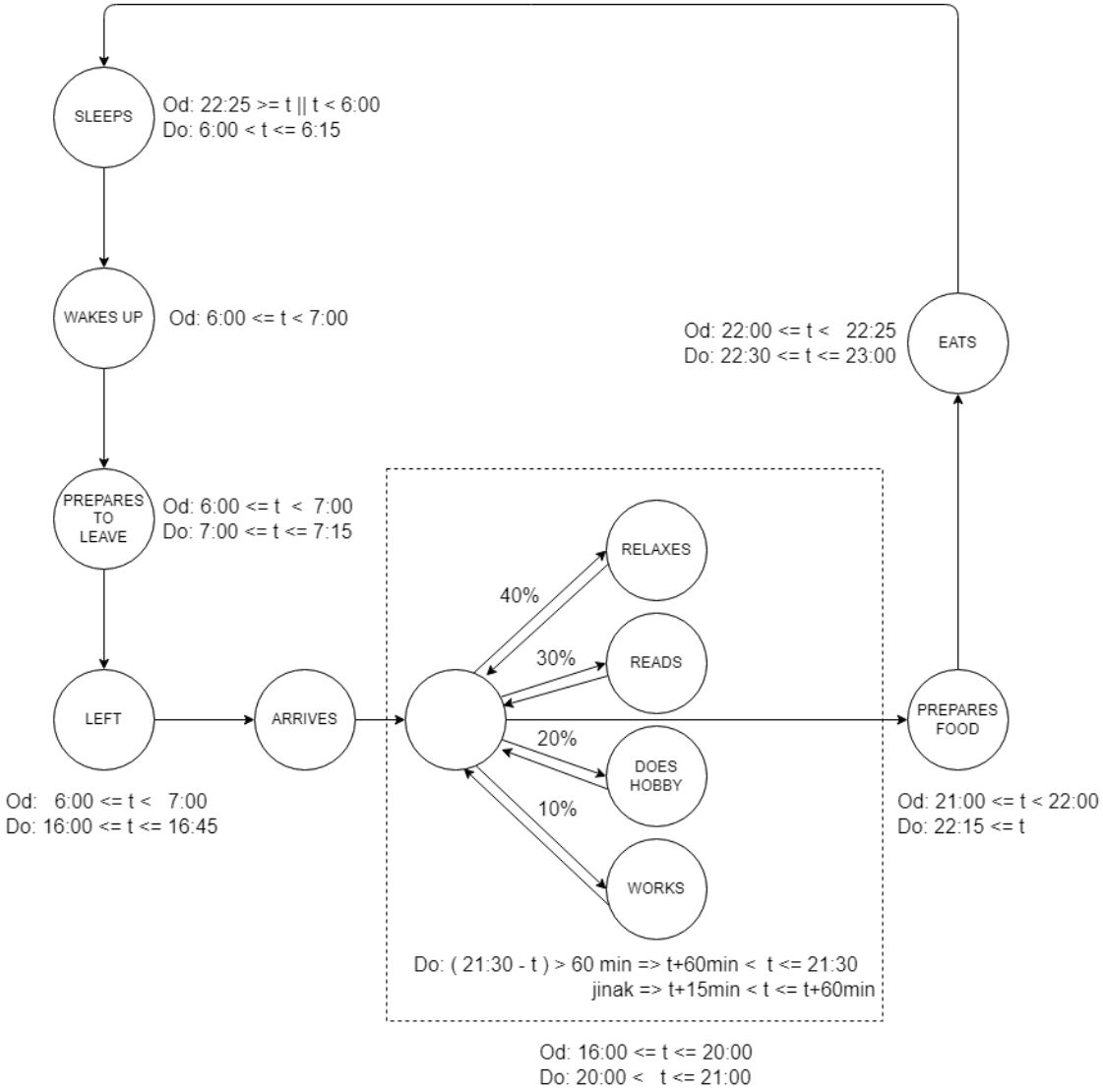
Tabulka 4.8: Stavy uživatelského modelu

Deterministický model

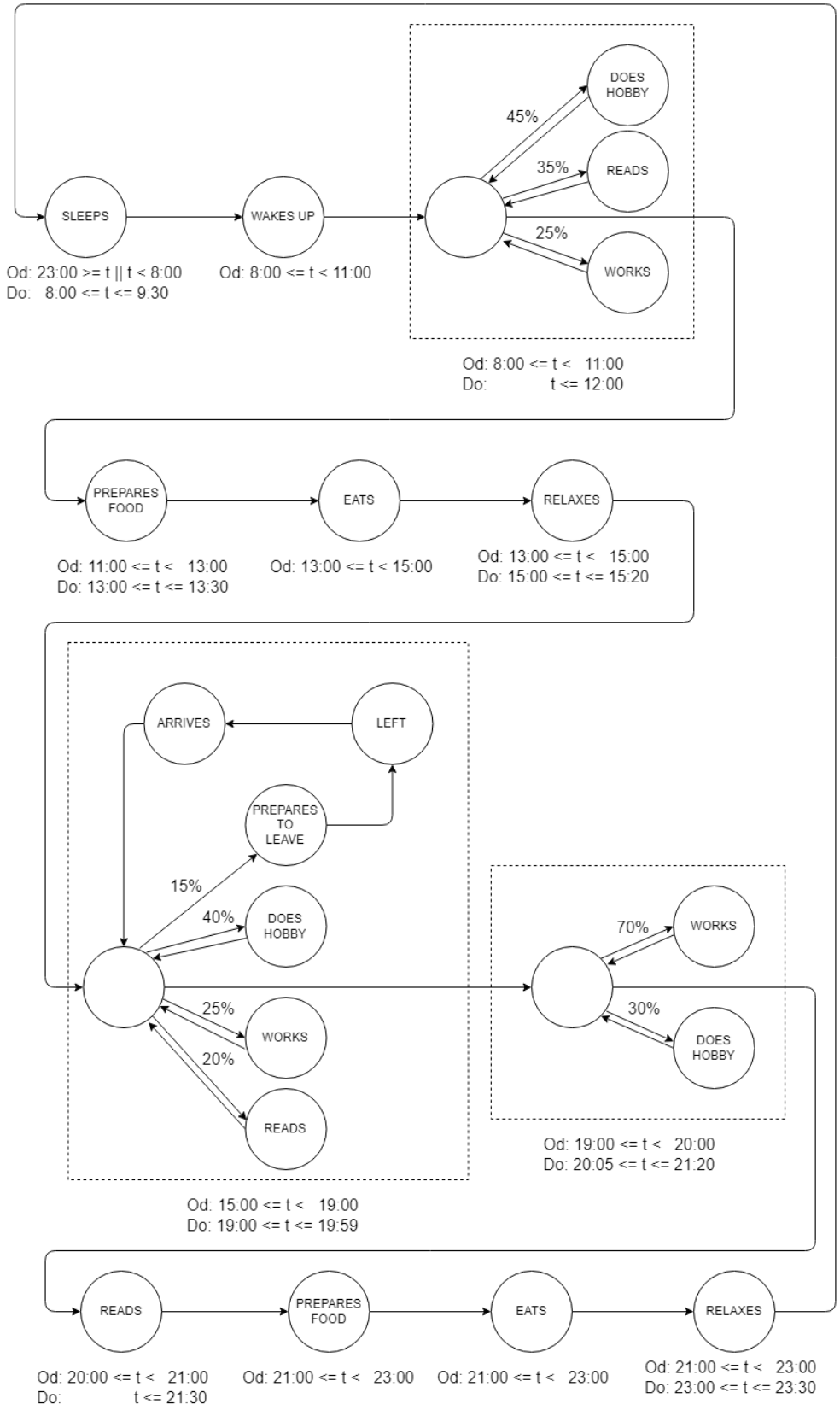
Model s deterministickým chováním uživatele slouží k ověření správnosti experimentů s neuronovými sítěmi, ve kterých by alespoň některé neuronové sítě měly mít dobré výsledky. Model je obdobný se stochastickým modelem s tím rozdílem, že každý stav trvá pokaždé pevně stanovenou dobu a rozhodování mezi více stavy podle pravděpodobnosti bylo nahrazeno pevnou sekvencí některých z možných stavů.

4.3.3 Technologie

Pro implementaci simulace jsem zvolil knihovnu *SimPy* [21]. SimPy je simulační knihovna pro simulace diskrétních událostí v jazyce Python. V SimPy lze jednoduše definovat procesy, události a simulační prostředí. Proces pak bude představovat model uživatele a simulační prostředí bude představovat Smart Home s chytrými zařízeními.



Obrázek 4.3: Chování uživatele v pracovní den



Obrázek 4.4: Chování uživatele o víkendu

Kapitola 5

Multiagentní systém

Tato kapitola se zabývá implementací multiagentního systému tak, jak je navržen v sekci 4.1, jeho konfigurací a použití skrze rozhraní, které lze integrovat do existujícího řídícího systému Smart Home. Systém je implementován ve složce `MAS-Simulation/MAS/` a v této kapitole se odkazuji na soubory/cesty v této složce s následující strukturou:

```
MAS
├── messages
│   ├── interfaceMessages.py
│   └── spadeMessages.py
├── models
│   ├── modelBase.py
│   ├── CfC.py
│   ├── FC.py
│   ├── LSTM.py
│   └── RNN.py
├── agents.py
├── config.py
├── data.py
├── interface.py
├── predictionModel.py
├── system.py
└── systemStats.py
```

SPADE používá pro komunikaci protokol XMPP a proto je pro fungování agentů nutné, aby se připojili k nějakému XMPP serveru (služba *jabber*)[1] a zaregistrovali se u něj. Každý agent je pak identifikován svým *jabber ID/JID* (např. *mainagent@localhost*), kterým se na XMPP serveru registroval. Tento identifikátor je pak používán k identifikaci odesílatele a příjemce zpráv.

Aby bylo možné s multiagentním systémem interagovat z vnějšího řídícího systému nebo ho při ověření zapojit do simulace, je nutné vytvořit rozhraní, které by bylo možné do vnějšího řídícího systému integrovat (např. v případě Home Assistant jako rozšíření/plugin). Rozhraní existuje mimo SPADE a pro zprostředkování komunikace skrze XMPP server jsem využil knihovnu *xmpppy*[2].

[1] jabber – https://jabber.org/
[2] xmpppy – https://xmpppy.sourceforge.net/

5.1 Agenti

V rámci SPADE lze celkové chování agenta definovat jako třídy jednotlivých dílčích chování, jejichž instance jsou agentovi přidány při jeho spuštění (v metodě `setup()`) nebo později při běhu jiného chování. Tyto třídy definující chování musí dědit z jedné ze základních tříd, z modulu `spade.behaviour`, podle periodicity jejich spouštění: `OneShotBehaviour` (spuštěné jen jednou), `CyclicBehaviour` (spouštěné vždy po dokončení), `PeriodicBehaviour` (spouštěné vždy s prodlevou po dokončení). Při ukončování agenta lze ještě provést poslední akce v metodě `stop()` a následně se agent ukončí. Dalšími důležitými metodami agenta jsou metody `set()` a `get()`, které umožňují agentovi si uložit/získat data do/z jeho báze znalostí. Interně je báze znalostí objekt typu `Dict` (*Dictionary*) a lze tak do ní uložit dvojici klíč, hodnota. Báze znalostí slouží nejen jako paměť agenta, ale také jako způsob, jak sdílet data napříč jednotlivými chováními. Agenti jsou implementováni v souboru `agents.py`.

Základní chování každého agenta je definováno ve třídě `BaseAgent`, která dědí ze třídy `spade.agent.Agent`. Tato třída implementuje cyklické chování `StopMessageBehavior`, ve kterém naslouchá zprávě pro zastavení agenta `StopMessage` a při její přijetí se agent zastaví (metoda `stop()`).

Predikční agenti a agenti uživatelů potřebují pro predikce a akce získat aktuální čas. Aby byl čas správný i při běhu systému v simulaci, lze při startu systému předat funkci, která se použije jako `get_time()` metoda v souboru `agents.py`.

5.1.1 Hlavní agent

Hlavní agent je implementován ve třídě `MainAgent`. Hlavní agent při spouštění nastaví svůj *JID* do globální proměnné, aby mu každý agent mohl zasílat zprávy. Do své báze znalostí si uloží dva prázdné seznamy `user_agents` a `device_agents`, do kterých si později ukládá přidávané predikční a uživatelské agenty. Dále inicializuje všechna svá chování pro příjem jednotlivých zpráv. Poslední přidané chování `SendReadyMessageBehavior` odešle rozhraní systému `AgentReadyMessage`, kterou oznamuje, že je systém spuštěný a připravený.

Přidávání agentů

V chováních `ReceiveAddNewDeviceAgentBehavior` a `ReceiveAddNewUserAgentBehavior` naslouchá hlavní agent zprávám z rozhraní systému s požadavky na přidání nových predikčních a uživatelských agentů do systému. Tyto zprávy obsahují přihlašovací údaje nového agenta, kterými se registruje na XMPP serveru (JID a heslo). Hlavní agent zkontroluje, zda již agent se stejným JID v systému neexistuje. Pokud ne, vytvoří nového predikčního nebo uživatelského agenta, přidá ho do seznamu agentů v systému, který si pamatuje ve své bázi znalostí, a nového agenta spustí. Nakonec hlavní agent zašle zprávu rozhraní systému oznamující úspěch (`SuccessMessage`) nebo neúspěch (`ErrorMessage`) podle toho, zda se povedlo agenta vytvořit a spustit.

Přeposílání zpráv

V chováních `ReceiveNewStateBehavior` a `ReceiveTriggerPredictionBehavior` hlavní agent naslouchá zprávám obsahujícím nový stav prostředí (`NewStateMessage`) a zprávám obsahujícím požadavek na novou predikci na vyžádání (`TriggerPredictionMessage`). Tyto zprávy přeposílá všem predikčním agentům a následně rozhraní systému odpoví zprávou o úspěchu (`SuccessMessage`). V chování `ReceivePredictionBehavior` hlavní agent

naslouchá zprávám s predikcí (`PredictionMessage`) od všech predikčních agentů. Tyto zprávy pak přeposílá všem uživatelským agentům k filtraci a interpretaci. Nakonec v chování `ReceiveActionBehavior` hlavní agent naslouchá zprávám (`ActionMessage`) od uživatelských agentů, které obsahují interpretovanou akci na základě předchozí predikce. Tyto zprávy pak přeposílá rozhraní systému.

Ukončení agenta

Při ukončování hlavního agenta ukončí nejprve hlavní agent všechny agenty v systému, které má uložené ve své bázi znalostí. Až po ukončení všech ostatních agentů se ukončí samotný hlavní agent a tím tak celý systém.

5.1.2 Predikční agent

Predikční agent je implementován ve třídě `DeviceAgent`. Při svém spouštění přidá chování pro naslouchání zpráv, ale nepřidá chování pro periodické predikce. To je přidáno až po přijetí prvního stavu prostředí, na základě kterého je inicializován predikční model.

Nové stavy prostředí

Naslouchání zprávám s novým stavem prostředí je implementováno v periodickém chování `ReceiveNewStateBehavior`. Z nového příchozího stavu si agent vyjme lokace uživatelů a stav zařízení, jehož stav predikuje. Pokud jde o první stav, který agent obdržel, inicializuje nový predikční model podle konfigurace systému, případně načte uložený model. Dále, podle konfigurace, si přidá chování pro periodickou predikci `PeriodicPredictionsBehavior` anebo chování pro jednorázovou predikci (predikce při obdržení stavu) `PredictBehavior`.

Predikční model

Predikční model je implementován třídou `PredictionModel` v souboru `predictionModel.py`. Při inicializaci vytvoří neuronovou síť, jejíž typ a parametry určuje konfigurace systému. Predikce je realizována v metodě `predict()`, ve které si predikční model predikci a vnitřní stav zapamatuje. Predikce a vnitřní stav jsou později využity k učícímu kroku v metodě `learn()`. Vnitřní stav je také využit v budoucích predikcích. Predikční model také implementuje ukládání a načítání naučeného modelu do/ze souboru v metodách `save()` a `load()`.

Konkrétní neuronové sítě jsou implementovány pomocí knihovny PyTorch ve složce `models/`. Každá ze tříd pro konkrétní neuronovou síť dědí ze společné bázové třídy `ModelBase` definované v souboru `modelBase.py`. Ta definuje základní rozhraní každého modelu, konkrétně inicializaci a metody `get_hparams()` a `forward()`. Konkrétní neuronové sítě jsou pak implementovány v ostatních souborech ve složce `models/`.

Predikce a učení

Jednorázová predikce, v chování `PredictBehavior`, je spouštěna z periodického chování `PeriodicPredictionsBehavior` nebo z chování `ReceiveTriggerPredictionBehavior`, které naslouchá zprávám `TriggerPredictionMessage` pro predikci na vyžádání. Pro predikci agent použije poslední známý stav prostředí. Před tím, než provede predikci, provede agent učící krok predikčního modelu na základě poslední predikce a aktuálního stavu prostředí (posledního známého). Následně agent provede predikci a zašle ji hlavnímu agentovi zprávou `PredictionMessage`.

5.1.3 Uživatelský agent

Predikční agent je implementován ve třídě `UserAgent` a při svém spouštění si přidá chování na přijímání zpráv `ReceiveSetDeviceFilterBehavior` a `RecievePredictionBehavior`.

Nastavení filtrace a interpretace

V chování `ReceiveSetDeviceFilterBehavior`, při obdržení zprávy pro nastavení filtru `SetDeviceFilterMessage`, si agent přidá do své báze znalostí nový filtr (`DeviceFilter`) pro nové zařízení, případně aktualizuje hodnoty již existujícího filtru pro dané zařízení. Zařízení rozlišuje podle JID predikčního agenta. Pokud příchozí filtr pro nové zařízení neobsahuje všechny hodnoty, jsou použity následující výchozí hodnoty:

- Povolení (`Enabled`) – `False`
- Hraniční hodnota vypnutí (`Threshold_Off`) – 0.5
- Hraniční hodnota zapnutí (`Threshold_On`) – 0.5

Pokud příchozí filtr je pro zařízení, pro které je filtr již nastaven, jsou aktualizovány pouze hodnoty, které příchozí filtr obsahuje. Nastavený filtr si agent uchová ve své bázi znalostí a odešle zprávu `SuccessMessage`.

Filtrace a interpretace

Filtrování a interpretování predikcí je realizováno v chování `RecievePredictionBehavior`, ve kterém agent čeká na zprávu s predikcí (`PredictionMessage`). Příchozí zpráva obsahuje JID predikčního agenta a samotnou predikci. Pokud uživatelský agent nemá ve své bázi znalostí nastavený filtr s JID predikčního agenta, zařízení není filtrem povoleno (`DeviceFilter.Enabled`) nebo je predikce mimo nastavené hraniční hodnoty filtru (`DeviceFilter.Threshold_Off` a `DeviceFilter.Threshold_On`), predikci ignoruje. Jinak agent pošle zprávu s akcí, resp. požadovaným stavem zařízení (`ActionMessage`).

5.2 Zprávy

Zprávy jsou implementovány jako šablony, které slouží k rozpoznávání příchozích zpráv a vytváření nových zpráv. Protože agenti posílají zprávy skrze knihovnu SPADE (třída `spade.message.Message`), ale vnější rozhraní systému skrze jinou XMPP knihovnu (třída `xmpp.Message`), jsou tyto šablony pro rozhraní a SPADE agenty implementovány zvlášť v souborech `messages/spadeMessages.py` a `messages/interfaceMessages.py`.

5.2.1 Zprávy SPADE

Při přidávání chování agentovi metodou `add_behavior()` lze také předat instanci šablony zpráv, které má chování přijímat. Uvnitř chování pak lze zprávám, které odpovídají šabloně, naslouchat metodou `receive()`. K porovnání příchozí zprávy se šablonou slouží metoda šablony `match()`, která vrací `True/False`. Odesílání zpráv uvnitř chování je prováděno instanciací šablony zprávy, kterou chce agent odeslat, doplněním příjemce a obsahu zprávy a odesláním skrze metodu `send()`.

Každá zpráva dědí z třídy `spade.message.Message`. Ta obsahuje základní definici metody `match()`, která porovnává všechny vyplněné atributy šablony. Nejdůležitějšími atributy jsou metadata, pomocí kterých lze šablonám přidat atributy z FIPA ACL. Díky těmto metadatům a jejich porovnáváním SPADE přímo FIPA ACL podporuje. Každá šablona zprávy má po instanciaci nastavené FIPA ACL atributy tak, jak jsou navrženy v sekci 4.1. Některé zprávy mají metodu `match()` rozšířenou o validaci formátu obsahu zprávy.

Aby agent při odesílání nastavil obsah zprávy ve správném formátu nebo při přijetí zprávy získal data v obsahu zprávy jako datovou strukturu, obsahují některé zprávy *property*, která převádí obsah zprávy na datovou strukturu a naopak. Datové struktury, včetně jejich serializace do formátu *JSON* pro zprávy s FIPA ACL atributem *language* = *json*, jsou definovány v souboru `data.py`.

Objekt, který vrací metoda `receive()`, je typu `spade.message.Message` a nebude tak tento objekt zprávy obsahovat *property* ani případné metody konkrétní zprávy. Proto každá třída zprávy obsahuje metodu `from_message()`, která z objektu přijaté zprávy vytvoří zprávu jako instanci třídy konkrétní zprávy.

5.2.2 Zprávy rozhraní

Společná bázová třída pro tyto zprávy `Message` dědí z `xmpp.Message`. Tato třída má metody pro přidávání a získávání SPADE metadat (atributů z FIPA ACL). Dalšími metodami jsou metoda `match()`, která obdobně jako u SPADE zpráv slouží k rozpoznání příchozí zprávy podle FIPA ACL atributů, a `expect_reply()` pro přidání *reply-with* FIPA ACL atributu.

Aby SPADE správně interpretoval metadata zpráv zaslané z rozhraní, je třeba tyto metadata vložit do zpráv tak, jak je SPADE očekává ve svých vlastních zprávách. Zprávy zasílané protokolem XMPP, tedy i SPADE zprávy, jsou ve formátu XML s protokolem definovanou strukturou. Metody `get_spade_metadata()` a `set_spade_metadata()` bázové třídy `Message` tuto XML strukturu prochází a správně metadata získá/nastaví. Tímto způsobem je zajištěna interoperabilita mezi SPADE zprávami a vnějšími XMPP zprávami rozhraní. Příklad, jak je SPADE zpráva strukturovaná, je v komentáři v souboru `messages/interfaceMessages.py`.

Každá konkrétní zpráva, dědící z bázové třídy `Message`, pak obdobně jako SPADE zprávy, nastavuje navržené FIPA ACL atributy jako SPADE metadata, obsahuje *property* pro získání/nastavení obsahu zprávy a případně implementuje vlastní `match()` metodu s validací dat.

Jedním z důležitých FIPA ACL atributů je i atribut *reply-with*, který rozhraní využívá k přiřazení odpovědi k původní odeslané zprávě. Hodnota tohoto atributu je *ID* odesílané zprávy, které je interně generováno knihovnou *xmpppy*.

5.3 Spuštění a používání systému

Tato podkapitola popisuje možnosti konfigurace, způsob spuštění a interagování s multiagentním systémem skrze rozhraní. Před spuštěním systému je třeba mít spuštěný XMPP server. Během testování jsem využil implementaci *ejabberd*[3] spuštěnou v Docker[4] kontejneru[5] ve verzi 24.02 s konfigurací specifikovanou v souboru `MAS-Simulation/ejabberd.yml`. Rozhraní je implementováno ve třídě `Interface` v souboru `interface.py` a spustitelná demon-

[3] ejabberd – https://www.ejabberd.im/
[4] Docker – https://www.docker.com/
[5] ejabberd docker kontejner – https://github.com/processone/ejabberd/pkgs/container/ejabberd

strace multiagentního systému a jeho použití s rozhraním je implementována v souboru `MAS-Simulation/InterfaceTest.py`.

5.3.1 Konfigurace

Multiagentní systém lze nakonfigurovat v souboru `MAS-Simulation/config.yaml`. Konfigurační soubor je ve formátu YAML[6] a obsahuje také konfiguraci simulace. Konfigurace multiagentního systému je v části `MAS:`, která se dělí na dvě další části. První část `prediction:` konfiguruje predikci a predikční model predikčních agentů a druhá část `logging:` konfiguruje výstupní záznamy. Konfigurace výstupních záznamů přímo odpovídá konfiguraci Python modulu `logging`[7]. Pro predikční model lze konfigurovat následující hodnoty:

- *prediction:* ..... Konfigurace predikce
  - *model_params:* ..... Konfigurace neuronové sítě
    - *type:* ..... Typ neuronové sítě (*CfC*, *LSTM*, *RNN*, *FC*)
    - *hidden_size:* ..... Velikost skrytých vrstev, u CfC počet neuronů
    - *num_layers:* ..... Počet vrstev, u CfC nepoužito
    - *learning_rate:* ..... Učící krok
    - *sequence_length:* ..... Délka vstupní sekvence
    - *keep_hidden_state:* ..... Zachování vnitřního stavu
  - *load_model:* ..... Použít uložené naučené modely
  - *save_model:* ..... Ukládání naučených modelů
  - *save_after_n_learning_steps:* . Uložit model každých *n* učících kroků
  - *models_folder:* ..... Složka uložených modelů
  - *predict_on_new_state:* ..... Generovat predikce při novém stavu
  - *periodic_prediction:* ..... Generovat predikce periodicky
    - *enabled:* ..... (Ne)povoleno
    - *period:* ..... Perioda v minutách

5.3.2 Spuštění

Při instanciaci rozhraní vnějším řídícím systémem jsou rozhraní předány parametry, kterými se rozhraní a později hlavní agent připojí k XMPP serveru: JID rozhraní, heslo rozhraní, JID hlavního agenta a heslo hlavního agenta. V případě, že by účty na XMPP serveru neexistovaly, rozhraní a hlavní agent se automaticky zaregistrují.

Rozhraní dále spustí obsluhu příchozích zpráv `_handle_message()` na jiném vlákně, která uloží zprávy s atributem *in-reply-to* pro hlavní vlákno, které na tuto zprávu čeká. Obsluha dále zpracovává příchozí zprávy na základě `match()` metod jednotlivých šablon zpráv.

Po inicializaci a připojení rozhraní je možné systém spustit. Spuštění systému realizuje metoda `start()`, která systém spustí jako podproces a předá mu všechny potřebné parametry:

- JID rozhraní
- JID hlavního agenta

[6] YAML – https://yaml.org/
[7] Python logging – https://docs.python.org/3/library/logging.html

- heslo hlavního agenta
- konfigurace výstupních záznamů a predikce
- funkce pro získání aktuálního času

Rozhraní pak čeká po určitou dobu (parametr `timeout`) než se systém spustí, případně vyvolá výjimku. Vstupním bodem systému je metoda `system_start()` v souboru `system.py`. Tato metoda nastaví konfigurace, funkci pro získání času a spustí SPADE s metodou `main()`. V metodě `main()` se vytvoří a spustí hlavní agent a po spuštění agenta metoda čeká na jeho ukončení. Hlavní agent po spuštění odešle zprávu `AgentReadyMessage` rozhraní a tím ho informuje, že multiagentní systém byl úspěšně spuštěn.

5.3.3 Přidávání agentů

Po spuštění multiagentního systému lze skrze rozhraní přidávat uživatelské a predikční agenty pomocí metod `add_user()` a `add_device()`. Těmto dvěma metodám jsou skrze parametry předány JID a heslo nového agenta.

Při přidávání uživatelského agenta je možné specifikovat jeho počáteční/aktuální lokaci. Pokud je lokace nového uživatele specifikována, bude pak součástí stavu prostředí a použita pro predikci. Protože nelze změnit vstupní vektor neuronových sítí, které se již učí, lze lokaci uživatele použít pouze v případě, kdy do systému ještě nebylo přidáno žádné zařízení. Při přidávání predikčního agenta je kromě JID a hesla pro nového agenta ještě předáván počáteční stav zařízení.

Rozhraní pak zašle zprávu `AddNewUserAgentMessage` nebo `AddNewDeviceAgentMessage` s JID a heslem hlavnímu agentovi a čeká na jeho odpověď v podobě zpráv `SuccessMessage` nebo `ErrorMessage`.

5.3.4 Nastavení uživatelských filtrů

Uživatelský filtr lze nastavit nebo aktualizovat metodou `user_set_device_filter()`, které jsou parametry předány JID uživatelského agenta, kterému filtr nastavit, JID predikčního agenta, kterého se filtr týká a parametry samotného filtru. Rozhraní zašle hlavnímu agentovi zprávu `SetDeviceFilterMessage` a čeká na jeho odpověď v podobě zpráv `SuccessMessage` nebo `ErrorMessage`.

5.3.5 Aktualizace stavu prostředí

Stav prostředí lze aktualizovat metodou `update_state()`, které jsou v parametrech předány lokace uživatelů a stavy zařízení jako dvojice JID uživatelského agenta a lokace nebo JID predikčního agenta a stavu. Není nutné metodě předat všechny lokace a stavy zařízení, ale jen ty, které se změnily. Rozhraní pak hlavnímu agentovi zašle celý stav prostředí s aktualizovanými hodnotami zprávou `NewStateMessage`. Další metodou pro aktualizaci stavu prostředí je metoda `update_state_with_response()`, která funguje obdobně, ale ještě počká na odpověď hlavního agenta. Ta je důležitá později, když je systém spuštěn v simulaci, aby simulace počkala, než proběhne aktualizace stavu.

5.3.6 Získání akcí

Pokud uživatelský agent na základě predikce vygeneruje akci, je tato akce skrze hlavního agenta zaslána rozhraní v podobě zprávy `ActionMessage`. Ta je přijata obsluhou pro pří-

chozí zprávy na rozhraní, kde je zařazena do fronty akcí. Z této fronty ji pak vnější řídící systém může získat metodou `pending_actions_pop()`, která vrátí nejstarší akci ve frontě.

Pro generování akcí na vyžádání lze využít metodu `trigger_predictions()`, která hlavnímu agentovi a ten pak predikčním agentům odešle zprávu `TriggerPredictionMessage`. Tato zpráva vynutí vygenerování predikcí pro všechna zařízení a uživatelští agenti pak na základě těchto predikcí mohou vygenerovat a zaslat akci. Případně lze využít metodu `trigger_predictions_with_response()`, která navíc počká na odpověď hlavního agenta oznamující, že všechny predikce a případné akce byly vygenerovány.

5.3.7 Zastavení systému

Systém lze zastavit metodou `stop()`, která hlavnímu agentovi zašle zprávu `StopMessage` a ten celý multiagentní systém ukončí. Rozhraní pak počká na vypnutí multiagentního systému a tím celého podprocesu. Rozhraní případně po určité době čekání podproces terminuje. Systém je tímto způsobem zastaven i při destrukci instance rozhraní (metoda `__del__()`).

Kapitola 6

Simulace

Tato kapitola se zabývá implementací simulačního prostředí tak, jak je navrženo v sekci 4.3, její konfigurací, generováním datové sady pro experimenty s neuronovými sítěmi a zapojením multiagentního systému do simulace. Implementace simulace se nachází ve složce `MAS-Simulation/Simulation/` a v této kapitole se odkazuji na soubory/cesty v této složce s následující strukturou:

```
Simulation
├── inhabitants
│   ├── deterministic.py
│   └── stochastic.py
├── constants.py
├── deviceModels.py
├── environment.py
├── event.py
├── homeModel.py
├── inhabitantModel.py
├── stateLogger.py
└── utils.py
```

Knihovna SimPy obsahuje několik základních objektů pro simulaci diskrétních událostí. Simulační prostředí (třída `simpy.Environment`) zastřešuje celý běh simulace a stará se o vkládání a vyjímání událostí (třída `simpy.Event`) do/z fronty a posouvání simulačního času. Procesy (třída `simpy.Process`), které modelují aktivní entity generující události v simulačním prostředí jako Python generátory/korutiny[1].

6.1 Modely a procesy

V této podkapitole jsou popsány jednotlivé modely a procesy v simulaci, které mezi sebou interagují a definují tak běh celé simulace. Nejprve je zde popsáno simulační prostředí, které obsahuje všechny objekty v simulaci. Dále jsou zde popsány modely Smart Home, uživatelů a proces zaznamenávání.

[1] Python generátory/korutiny – speciální metody, které vracejí iterovatelný objekt, který generuje nové hodnoty klíčovým slovem `yield`. https://www.geeksforgeeks.org/generators-in-python/

6.1.1 Prostředí

Simulační prostředí je implementováno ve třídě `Environment` v souboru `environment.py` a dědí z třídy `simpy.Environment`. Základní SimPy prostředí rozšiřuje o model Smart Home, zasílání a naslouchání událostem (události mimo SimPy), jinou reprezentaci času a nové SimPy události.

Simulační čas v SimPy je reprezentován jen jako číslo bez dalšího významu. Aby šlo s časem lépe pracovat v kontextu chování uživatele, je čas reprezentován datovou strukturou `Time` obsahující několik hodnot: minuty, hodiny, dny, měsíce a roky. Čas v SimPy jsem se rozhodl interpretovat jako základní jednotku odpovídající minutám. Struktura pak implementuje převod mezi strukturovaným časem a časem v minutách a základní aritmetiku s touto strukturou.

Protože má navržený model uživatelského chování časová omezení, je třeba u každé SimPy události zkontrolovat, zda čas, na který by byla naplánována (vložena do fronty událostí), nepřesahuje čas, po který lze v daném stavu zůstat. Implementace SimPy událostí je naplánuje hned během inicializace události a nelze tak událost zkontrolovat. Proto je prostředí rozšířeno o novou událost `TimeoutRequest`, která se naplánuje až metodou `confirm()` a lze ji tak ještě před naplánováním zkontrolovat.

6.1.2 Smart Home

Smart Home je modelován třídou `Home` v souboru `homeModel.py`. Tato třída obsahuje místnosti a implementuje metody pro jejich přidání `add_room()`, pro vstup do místností `go_to_room()` a získání operace zařízení `get_device_op()`. Metoda `go_to_room()` vyvolá událost, že do dané místnosti vstoupil konkrétní uživatel, a danou místnost vrátí. Metoda `get_device_op()` vrátí výsledek stejnojmenné metody dané místnosti.

Místnost je modelována třídou `Room` v souboru `homeModel.py`. Tato třída obsahuje chytrá zařízení a implementuje metody pro jejich přidání `add_device()`, zpřístupnění `get_device()` a získání jejich operací `get_device_op()`. Každá místnost je identifikována svým názvem.

Zařízení je modelováno bázovou třídou `SmartDevice`, ze které dědí konkrétní zařízení. Bázová třída obsahuje mapování mezi názvem operace a metodou, která operaci implementuje, a statistiky o změnách stavu zařízení multiagentním systémem. Třída `SmartLight` představuje zařízení chytrého osvětlení a obsahuje operace/metody `turn_on()` a `turn_off()`, které pokud změní stav zařízení (mohlo být už zapnuto/vypnuto) vyvolá událost, že dané zařízení zapnul/vypnul konkrétní uživatel. Pro spínání multiagentním systémem obsahuje metody `MAS_turn_on()` a `MAS_turn_off()`, které událost nevyvolávají, ale slouží k počítání statistik pro dané zařízení.

6.1.3 Uživatel

Bázovou třídou pro všechny modely uživatelů je třída `Inhabitant` implementovaná v souboru `inhabitantModel.py`. Bázová třída obsahuje mapování mezi stavy uživatelského chování a metodami/generátory (metody `..._state()`, např. `eats_state()`), které chování v daných stavech implementují. Přecházení mezi stavy a nastavení jejich časových omezení je implementováno v metodách/generátorech `workday_behavior()` a `weekend_behavior()`, které odpovídají přecházení mezi stavy v pracovní dny a o víkendu, a mohou generovat SimPy události v případě, že modelují i chování při přechodu mezi stavy. Časové omezení setrvání ve stavu nastavují tyto metody v atributu `self.stateEnd`. Metody pro přecházení

mezi stavy, `workday_behavior()` a `weekend_behavior()`, a metody pro jednotlivé stavy jsou v bázové třídě prázdné a je pak na konkrétních modelech, aby tyto metody implementovaly.

Celkové chování zastřešuje metoda/generátor `behavior()`, která je použita jako proces (třída `simpy.Process`) v simulaci. Tato metoda vybere mezi `workday_behavior()` a `weekend_behavior()` podle aktuálního simulačního času a generuje SimPy události z těchto dvou metod a z metody pro aktuální stav. Aktuální stav zastřešuje metoda/generátor `current_state_actions()`, která generuje SimPy události z metody pro aktuální stav a u každé této události (`TimeoutRequest`) zkontroluje, že neporuší časové omezení setrvání ve stavu nastavené v atributu `self.stateEnd`. V případě, že naplánování události neporuší časová omezení, je touto metodou naplánována (`TimeoutRequest.confirm()`) a propagována dále. Pokud by naplánování události přesáhlo maximální čas, je událost ignorována a metoda končí. Naopak pokud by aktuální stav skončil před minimálním časem, vygeneruje tato metoda SimPy událost tak, aby ho prodloužila. Je tak zaručeno, že každý příliš dlouho trvající stav je zkrácen a každý příliš krátký stav je prodloužen.

Stochastický uživatel

Model stochastického uživatele je implementován v souboru `inhabitants/stochastic.py` třídou `StochasticInhabitant`. Metody `workday_behavior()` a `weekend_behavior()` jsou implementovány tak, aby odpovídaly navrženému modelu v sekci 4.3.

Metody implementující chování v jednotlivých stavech typicky nejprve přejdou do místnosti, kde uživatel vykonává danou aktivitu, a vygenerují SimPy událost simulující čas potřebný k přesunu do dané místnosti. Většina aktivit v místnosti je pak simulována rozsvícením světla, vygenerováním SimPy události simulující dobu trvání aktivity a zhasnutím světla po vykonání aktivity. Daný stav může v místnosti vykonat více takových aktivit opakováním této posloupnosti. Simulované doby trvání jsou náhodné v určitém intervalu a dané exponenciálním rozložením. Některé stavy spoléhají na jejich prodloužení nebo zkrácení mechanismem v bázové třídě `Inhabitant`.

Deterministický uživatel

Model deterministického uživatele je implementován třídou `DeterministicInhabitant` v souboru `inhabitants/deterministic.py`. Metody implementující přechody mezi stavy, `workday_behavior()` a `weekend_behavior()`, jsou úpravou metod stochastického uživatele tak, že každé náhodné rozhodování mezi stavy bylo nahrazeno deterministickou sekvencí některých z nich. Metody implementující chování v jednotlivých stavech jsou pak také úpravou metod stochastického uživatele tak, že náhodné doby trvání byly nahrazeny pevnými dobami trvání. Těmito náhradami náhodných prvků stochastického uživatele za prvky pevně dané je pak zaručeno deterministické chování.

6.1.4 Zaznamenávání

Zaznamenávání v simulaci slouží k tvorbě výstupních trénovacích dat pro experimenty s neuronovými sítěmi. V simulaci jsou implementovány dva způsoby zaznamenávání. Prvním způsobem je periodické zaznamenávání, které periodicky zaznamenává stav simulačního prostředí (prostředí Smart Home) s pevným časovým krokem, a jeho výstupem je tak časová řada vhodná pro experimenty. Druhým způsobem je zaznamenávání při každé změně stavu simulačního prostředí. Třídy implementující tyto dva způsoby obsahují metody, které jsou

při spuštění simulace zaregistrovány, aby naslouchaly jednotlivým událostem v simulaci. Zaznamenávání je implementováno v souboru `stateLogger.py`.

Periodické zaznamenávání je implementováno třídou `PeriodicStateLogger`. Tato třída si udržuje aktuální stav prostředí, který je definován třídou `State`. Třída obsahuje metodu/generátor `logBehavior()`, která slouží k vytvoření procesu v simulaci. Zaznamenávání při změně stavu prostředí pouze přímo reaguje na události a nemusí tak pro něj existovat proces v simulaci. Toto zaznamenávání je implementováno třídou `EventStateLogger`.

Výstupní data jsou zapsána do souborů ve formátu CSV[2] do nakonfigurované složky (*Simulation:inhabitant_logs:folder:*). Každý záznam/řádek obsahuje složky strukturovaného času, lokace uživatelů a stavy jednotlivých zařízení.

6.2 Běh simulace

V této podkapitole je popsáno použití a běh simulace a výstupní data, která simulace generuje. Simulaci lze spustit s nebo bez multiagentního systému. Bez multiagentního systému slouží simulace primárně k generování trénovacích dat pro experimenty s neuronovými sítěmi. S multiagentním systémem simulace slouží k jeho ověření.

6.2.1 Konfigurace

Běh simulace lze konfigurovat v souboru `MAS-Simulation/config.yaml` ve formátu YAML. Při spouštění simulačních programů lze skrze argument příkazového řádku předat cestu ke konfiguračnímu souboru. Soubor obsahuje jak konfiguraci multiagentního systému, tak simulace. Konfigurace multiagentního systému je popsána v podsekci 5.3.1. Konfigurace simulace je v části `Simulation:`, ve které lze konfigurovat následující hodnoty:

- *Simulation*: ..... Konfigurace simulace
  - *start*: ..... Počáteční čas simulace
    - *minute*: ..... Minuta
    - *hour*: ..... Hodina
    - *day*: ..... Den
    - *month*: ..... Měsíc
    - *year*: ..... Rok
  - *end*: ..... Konečný čas simulace
    - *minute*: ..... Minuta
    - *hour*: ..... Hodina
    - *day*: ..... Den
    - *month*: ..... Měsíc
    - *year*: ..... Rok
  - *log_interval*: ..... Interval logování výstupních dat
  - *MAS_update_interval*: ..... Interval aktualizace stavu pro MAS
  - *stochastic_inhabitants*: ..... Počet stochastických uživatelů
  - *deterministic_inhabitants*: ..... Počet deterministických uživatelů
  - *inhabitant_logs*: ..... Zaznamenávání výstupních dat
    - *enabled*: ..... (Ne)povoleno
    - *folder*: ..... Složka kde záznamy vytvořit
  - *jabber_host*: ..... Hostitel XMPP serveru

[2] CSV – https://www.geeksforgeeks.org/csv-file-format/

```
└─ prediction_filter: ..... Predikční filtr uživatelských agentů
   ├─ enabled: ..... (Ne)povoleno
   ├─ threshold_off: ..... Hraniční hodnota pro vypnutí
   └─ threshold_on: ..... Hraniční hodnota pro zapnutí
└─ TCP_relay_agent: ..... Nastavení TCP agenta
   ├─ host_ip: ..... IP adresa, na které naslouchá
   └─ host_port: ..... Port, na kterém naslouchá
```

6.2.2 Běh simulace bez MAS

Simulace bez MAS je implementována programem `MAS-Simulation/Simulation.py`. Program při spuštění vytvoří prostředí, modely uživatelů a nastaví zaznamenávání podle konfigurace. Uživatelé a periodické zaznamenávání jsou přidány do simulace jako proces. Simulace je pak spuštěna od počátečního do koncového času.

6.2.3 Popojení s MAS

Při propojení multiagentního systému se simulací skrze vnější rozhraní se ukázalo, že komunikace pomocí XMPP serveru je příliš pomalá pro simulaci dlouhých časových úseků. Dobu potřebnou pro zaslání nového stavu a čekání na akce zaslané systémem jsem experimentálně změřil a došel k hodnotě přibližně 2.6 sekundy. Jestliže by tato výměna probíhala každých 5 minut simulačního času, pak simulace 1 roku fungování systému by trvala cca 77.376 hodin. Je tak potřeba realizovat komunikaci jiným způsobem.

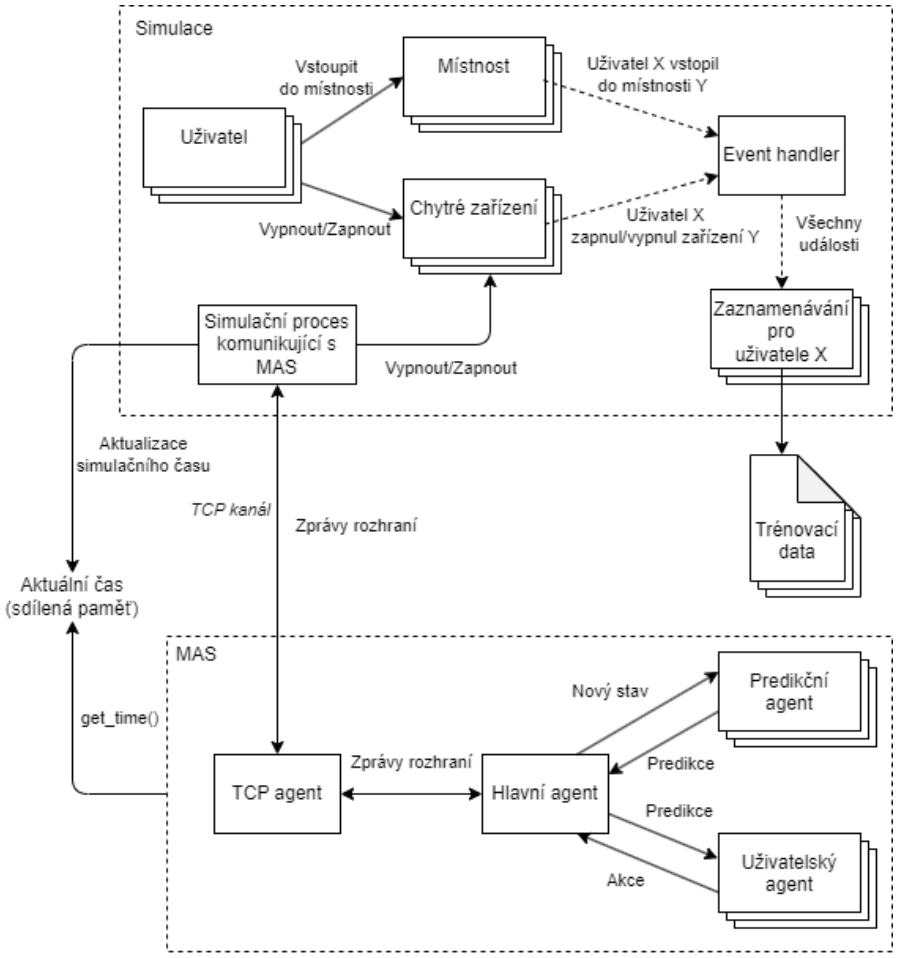
Komunikaci mezi simulací a systémem jsem vyřešil použitím TCP soketů. K tomuto účelu je v systému vytvořen nový agent `TCPRelayAgent`, který otevře TCP soket, čeká na navázání spojení (pouze jednoho) a v rámci tohoto spojení přeposílá příchozí/odchozí zprávy do/ze systému. JID tohoto agenta je nastaveno jako JID rozhraní, aby hlavní agent posílal zprávy tomuto agentu namísto rozhraní mimo systém. Pro serializaci zpráv skrze TCP spojení je použit jednoduchý protokol. Objekt představující zprávu je serializován pomocí modulu Pickle. Při zaslání zprávy jsou nejprve zaslány 4 byty určující délku zprávy a za nimi následuje samotná serializovaná zpráva. Přijímající strana pak nejprve čeká na příjem délky zprávy a následně na příjem celé zprávy, kterou deserializuje opět pomocí modulu Pickle. Zasílané zprávy jsou SPADE zprávy, aby bylo možné je uvnitř systému rovnou přeposlat příjemci.

Multiagentní systém je spuštěn v podprocesu obdobně jako ho spouští rozhraní. Aby multiagentní systém správně fungoval v simulačním čase, je mu předána funkce `get_time()`, která přečte simulační čas ze sdílené paměti mezi procesem simulace a podprocesem multiagentního systému.

6.2.4 Běh simulace s MAS

Simulace s připojeným multiagentním systémem je implementována programem v souboru `MAS-Simulation/SimulationMAS.py`. Program při spuštění vytvoří prostředí, modely uživatelů a nastaví zaznamenávání, tak jak při běhu bez MAS. V metodě `MAS_start()` spustí multiagentní systém v podprocesu s novým TCP agentem (`TCPRelayAgent`) metodou `system_start_tcp()` v souboru `MAS/system.py`, otevře TCP soket a opakovaně se zkouší připojit k soketu TCP agenta. Po úspěšném spuštění a připojení jsou systému poslány zprávy k přidání uživatelského agenta pro každého simulovaného uživatele, zprávy

k přidání predikčního agenta a nastavení uživatelských filtrů pro každé zařízení v simulaci. Přidávání agentů v tomto pořadí umožňuje predikčním modelům využít k predikci lokaci všech uživatelů. Před spuštěním simulace je do ní přidán SimPy proces pro obsluhu multiagentního systému realizovaný metodou `MAS_handling()` a následně je simulace spuštěna. Tato metoda, v intervalu specifikovaném v konfiguraci (`MAS_update_interval`), aktualizuje čas uložený ve sdílené paměti metodou `MAS_update_time()`, pošle systému zprávu s novým stavem a počká, až systém pošle akce pro všechna zařízení v simulaci, které vykoná. Každé zařízení, pro které systém zaslal akci k vykonání, započítá akci do statistik pouze pokud se stav zařízení změnil. Pokud po provedení systémové akce, která změní stav zařízení, provede některý z uživatelů interakci se zařízením, je systémová akce započítána jako správná, jestliže uživatelská interakce nezměnila stav zařízení. V opačném případě je započítána jako nesprávná. Po dokončení simulace je multiagentní systém ukončen a na standardní výstup jsou vypsány počty správných a nesprávných systémových akcí. Takto propojená simulace a multiagentní systém jsou znázorněny na diagramu 6.1.



Obrázek 6.1: Propojení simulace a multiagentního systému

Kapitola 7

Hluboké učení

Tato kapitola se zabývá implementací experimentů s neuronovými sítěmi za účelem ověření, že se multiagentní systém učí predikovat uživatelské interakce. Experimenty jsou implementovány ve složce `ModelTraining` a v této kapitole se odkazuji na soubory/cesty v této složce s následující strukturou:

```
ModelTraining
├── Components
│   ├── Data
│   │   └── SequentialData.py
│   ├── Metrics
│   │   └── StateChangeAccuracy.py
│   ├── Models
│   │   ├── CfC.py
│   │   ├── FC.py
│   │   ├── LSTM.py
│   │   ├── RNN.py
│   │   └── RNN.py
│   ├── DataModules.py
│   ├── LearnerModules.py
│   ├── ModelBase.py
│   └── ModelFactory.py
└── Lightning.py
```

7.1 Příprava dat

Vstupními daty pro experimenty jsou dvě datové sady ve formátu CSV generované simulací. První datová sada je použita pro trénování sítě a měla by být dostatečně dlouhá (např. jeden simulační rok). Druhá datová sada je použita pro zhodnocení (evaluaci), jak dobře je model naučený a může být kratší než trénovací datová sada (např. jeden simulační měsíc). Načtení datové sady z CSV souboru je implementováno třídou `SequentialStateDataset` v souboru `Components/Data/SequentialData.py`. Tato třída reprezentuje datovou sadu a využívá knihovnu *pandas*[1] pro načtení CSV souboru a knihovnu *NumPy*[2] pro mani-

[1] pandas – https://pandas.pydata.org/
[2] NumPy – https://numpy.org/

pulaci s načtenými daty. Načtená data lze indexovat metodou `__getitem__()`, která pro daný index vrátí sekvenci záznamů/stavů prostředí dlouhou podle parametru `lookback` jako vstup pro neuronovou síť (*features*) a následující stav prostředí, který by neuronová síť měla predikovat (*label*). Ve stejném souboru je také definována transformace jako třída `RemoveColumnsTransform`, kterou lze použít pro odstranění nechtěných sloupců. Pro experimenty je vhodné odstranit některé nepoužité hodnoty vstupní sekvence a v následujícím stavu prostředí ponechat jen stavy zařízení.

V PyTorch Lightning se pro načítání dat během experimentů používají datové moduly s bázovou třídou `LightningDataModule`, které obsahují několik datových sad, zejména trénovací a validační datové sady. Datový modul pro experimenty je implementován třídou `SequenceDataModule` v souboru `Components/DataModules.py`, která využívá datovou sadu `SequentialStateDataset` a transformaci `RemoveColumnsTransform` pro načtení datových sad z CSV souborů tak, aby odpovídaly návrhu v podsekci 4.2.1. Datový modul pak vzorky (vstupní sekvence a následující stav) z datových sad shlukuje do dávek podle konfigurace. Toto shlukování pak vede k rychlejším výpočtům během trénování. Datový modul navíc umožňuje dávky načítat paralelně pomocí několika vláken podle konfigurace.

7.2 Modely

Jednotlivé typy neuronových sítí jsou implementovány ve složce `Components/Models` pomocí knihovny PyTorch a v případě modelu CfC sítě je navíc využita implementace NCPs [13]. Oproti multiagentnímu systému, který vytváří model pro každé zařízení zvlášť, lze v PyTorch Lightning trénovat pouze jeden model najednou. Proto každý z těchto modelů uvnitř obsahuje modely pro každé zařízení zvlášť a pro PyTorch Lightning se tak jedná o jeden model. Při inicializaci je vytvořen model pro každé zařízení podle parametrů, včetně parametrů určujících velikost vstupu jako velikost společné části stavu prostředí a velikost částí stavu prostředí specifických pro každé zařízení. Metoda `forward()`, která vykoná dopředný průchod sítěmi, na základě těchto velikostí transformuje vstup na vstupy pro modely jednotlivých zařízení. Výstupy a skryté stavy modelů jsou pak transformovány a výstupem této metody jsou sjednocené výstupy a skryté stavy všech modelů. Další metodou je metoda `get_hparams()`, která vrací parametry (hyperparametry) modelu. Každý z modelů dědí z bázové třídy `ModelBase` v souboru `Components/ModelBase.py`, která definuje rozhraní pro tyto dvě metody, aby se s každým modelem dalo pracovat stejným způsobem. Pro usnadnění vytváření modelů podle konfigurace je ve třídě `ModelFactory` v souboru `Components/ModelFactory.py` implementována metoda `create_model()`, která podle parametrů vytvoří jeden z modelů. Implementovány jsou následující třídy modelů:

- `PerDeviceFC` – Model plně propojené sítě v souboru `Components/Models/FC.py`.
- `PerDeviceRNN` – Model RNN sítě v souboru `Components/Models/RNN.py`.
- `PerDeviceLSTM` – Model LSTM sítě v souboru `Components/Models/LSTM.py`.
- `PerDeviceCfC` – Model CfC sítě v souboru `Components/Models/CfC.py`.

7.3 Experimenty

Běh experimentu je implementován programem v souboru `Lightning.py`. Program podle konfigurace vytvoří datový modul, model neuronové sítě a trénovací modul a experiment spustí.

7.3.1 Konfigurace

Běh experimentu lze konfigurovat v souboru `config.yaml` ve formátu YAML. Konfigurace je rozdělena na několik částí. První část *dataset:* konfiguruje CSV soubory, které obsahují trénovací a validační datové sady, počet sloupců obsahujících hodnoty strukturovaného času, počet chytrých spínačů, počet uživatelů a časový krok mezi záznamy. Druhá část *datamodule:* konfiguruje velikost dávek a délku vstupní sekvence. Třetí část *model:* konfiguruje velikost neuronové sítě. Čtvrtá část *training:* konfiguruje způsob trénování a poslední část konfiguruje složku, kam ukládat výsledky experimentů.

```
config.yaml ..................................... Konfigurační soubor
├─ dataset: ................................... Konfigurace datových sad
│  ├─ train_path: ....................... Cesta k trénovací datové sadě
│  ├─ eval_path: ........................ Cesta k validační datové sadě
│  ├─ time_columns: ............................ Počet časových sloupců
│  ├─ n_smart_devices: ....................... Počet chytrých zařízení
│  ├─ n_users: ........................................ Počet uživatelů
│  └─ time_step: ............................. Velikost časového kroku
├─ datamodule: ............................ Konfigurace datového modulu
│  ├─ batch_size: ..................................... Velikost dávky
│  ├─ sequence_len: ........................... Délka vstupní sekvence
│  └─ num_workers: ..................... Počet vláken pro načítání dat
├─ model: ..................................... Konfigurace modelu
│  ├─ hidden_size: .... Velikost skrytých vrstev, pro CfC počet neuronů
│  └─ num_layers: ..................... Počet vrstev, nevyužito pro CfC
├─ training: ................................... Konfigurace trénování
│  ├─ keep_hidden_state: ..... Zachovat vnitřní stav mezi učícími kroky
│  ├─ max_epochs: ............................. Maximální počet epoch
│  └─ learning_rate: ....................... Velikost trénovacího kroku
└─ logging: ............................ Konfigurace výstupních záznamů
   └─ log_folder: ................. Složka kam výstupní záznamy ukládat
```

Při spouštění experimentu lze skrze argument příkazového řádku předat cestu ke konfiguračnímu souboru a další parametry, přičemž `-m`/`--model` je povinný:

- `-v`, `--version` VERZE – Verze/název experimentu.
- `-m`, `--model` MODEL – Typ neuronové sítě, s kterou experiment provést.
- `-c`, `--config` CESTA – Cesta ke konfiguračnímu souboru.
- Další argumenty, kterými lze přepsat hodnoty z konfigurace.

7.3.2 Trénovací modul

V PyTorch Lightning musí trénovaný model dědit ze třídy `LightningModule`. Tato třída poskytuje abstrakci nad procesem trénování, validace a zaznamenávání výstupů a metody, kterými lze definovat jednotlivé kroky trénování. Trénovací modul, implementován třídou `SequenceLearner` v souboru `LearnerModules.py`, dědí z této třídy a zapouzdřuje model neuronové sítě. Pro trénování používá ztrátovou funkci `MSELoss`[3] z modulu `torch.nn` a optimalizační algoritmus `Adam`[4] z modulu `torch.optim`. Proces trénování a validace jsou definovány, mimo jiné, následujícími metodami:

- `_base_step()` – Základní krok s dopřednou propagací, vypočítáním ztrátové funkce a uchováním skrytého stavu.
- `training_step()` – Trénovací krok. Využívá základní krok a zaznamenává metriky během učení.
- `validation_step()` – Validační krok. Využívá základní krok a zaznamenává metriky během validace.
- `on_validation_epoch_end()` – Zaznamenává hodnoty všech metrik na konci validace.

Predikce jsou omezeny na interval <0.0, 1.0> a interpretovány podle hraniční/prahové hodnoty 0.5, jinými slovy jsou zaokrouhleny. Hodnoty predikce pak odpovídají stavu zařízení (0 = vypnuto, 1 = zapnuto). Upravené hodnoty predikce, které jsou použity pro výpočet metrik spolu s opravdovým stavem zařízení, jsou uvedeny v tabulce 7.1.

Metrika	Vstup
Accuracy	Zaokrouhlená predikce
Precision	Zaokrouhlená predikce
Recall	Zaokrouhlená predikce
F1Score	Zaokrouhlená predikce
AUROC	Predikce omezena na interval
Hammingova vzdálenost	Zaokrouhlená predikce
Přesnost při změně stavu	Zaokrouhlená predikce

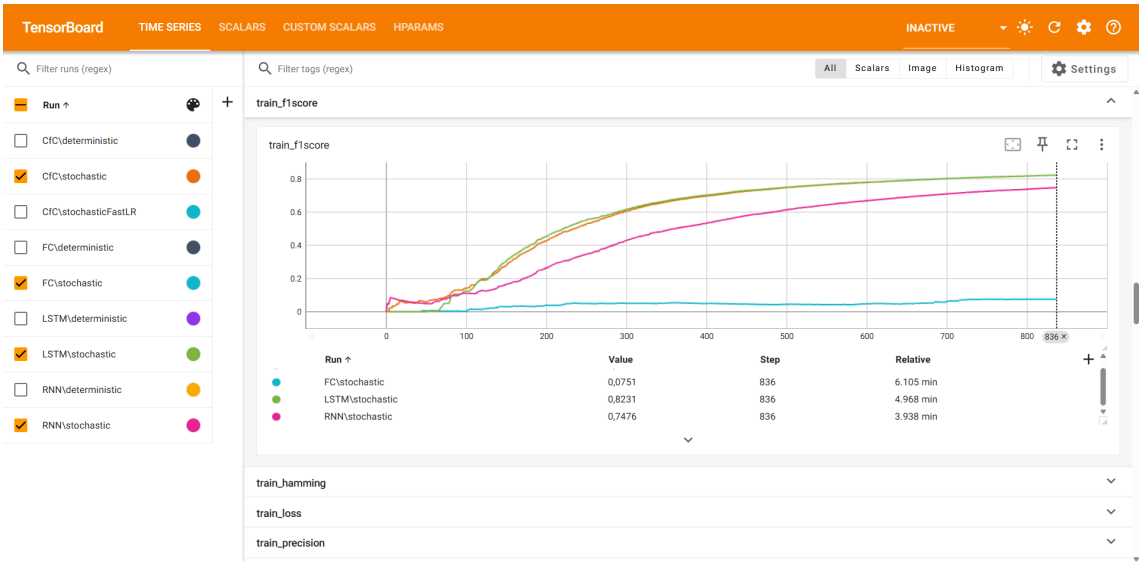
Tabulka 7.1: Stavy uživatelského modelu

7.4 Výstupní data

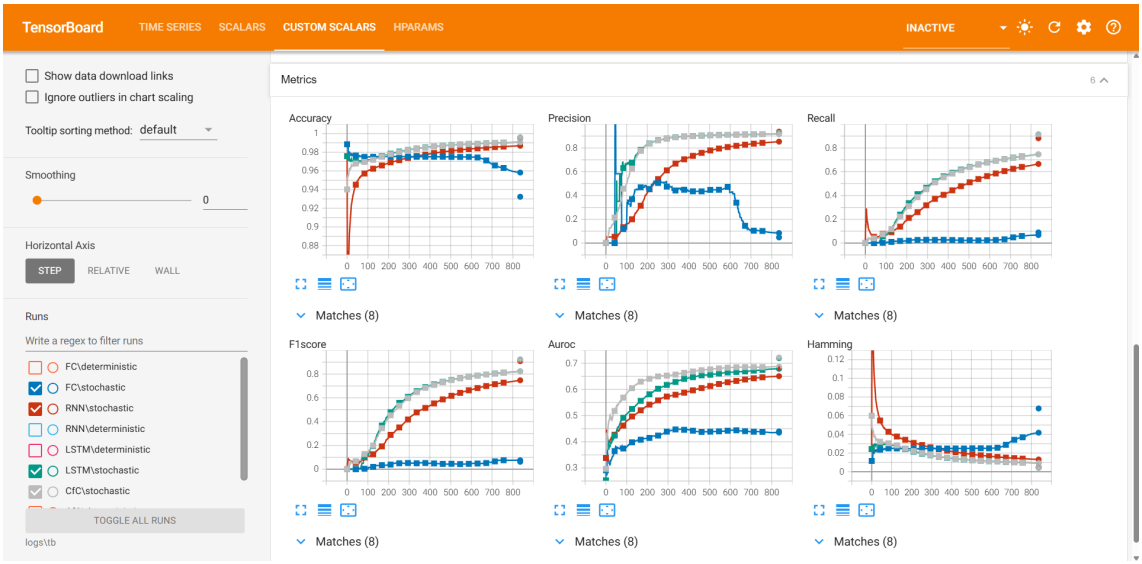
Výstupními daty experimentu jsou zaznamenávané průběhy a metriky v průběhu trénování a validace. Tato data pak lze vizualizovat nástrojem TensorBoard a slouží ke srovnání a ověření neuronových sítí. Výstupy jsou ukládány ve formátu CSV a formátu pro TensorBoard do nakonfigurované složky ve složkách podle formátu, názvu modelu a názvu/verze experimentu, např. `logs/tb/CfC/version_1` nebo `logs/csv/LSTM/version_2`. Trénovací modul během experimentu zaznamenává metriky navržené v podsekci 4.2.3 průběžně během trénování a na konci validace pro celkový predikovaný stav a pro predikce stavů jednotlivých

[3] `MSELoss` – https://docs.pytorch.org/docs/stable/generated/torch.nn.MSELoss.html
[4] `Adam` – https://docs.pytorch.org/docs/stable/generated/torch.optim.Adam.html

zařízení. Navíc během validace zaznamenává průběh stavu zařízení a jeho predikce pro každé zařízení. V nástroji TensorBoard jsou v části `TIME SERIES` vizualizovány všechny zaznamenané průběhy a hodnoty metrik na konci validace. Tato část je ukázána na obrázku 7.1. V části `CUSTOM SCALARS` jsou vizualizovány průběhy stavů zařízení s jejich predikcemi v jednom grafu pro každé zařízení a průběhy metrik během trénování s jejich výslednou hodnotou po validaci. Tato část je ukázána na obrázku 7.2. Tyto grafy jsou definovány trénovacím modulem v metodě `add_custom_scalers()`.



Obrázek 7.1: Ukázka obrazovky TIME SERIES v TensorBoard



Obrázek 7.2: Ukázka obrazovky CUSTOM SCALARS v TensorBoard

Kapitola 8

Ověření

Tato kapitola se zabývá ověřením funkčnosti implementovaného multiagentního systému, vyhodnocením výsledků a diskutuje možnost přechodu k reálnému nasazení. Schopnost učení je ověřena na základě experimentů s neuronovými sítěmi implementovaných v kapitole 7. Funkčnost multiagentního systému jako celku je ověřena jeho propojením s implementovaným simulačním prostředím.

8.1 Experimenty s neuronovými sítěmi

Cílem této sekce je ověřit schopnost neuronových sítí naučit se predikovat uživatelské akce. Experimenty byly provedeny s datovými sadami vygenerovanými simulačním prostředím implementovaným v kapitole 6. Ověření a srovnání neuronových sítí bylo provedeno na základě navržených a zaznamenávaných metrik.

8.1.1 Datové sady a parametry

Za účelem experimentů byly simulací vygenerovány dvě trénovací datové sady s pevným časovým krokem 5 minut po dobu 2 simulačních roků s lokací uživatele. První datová sada byla vygenerována simulací jednoho deterministického uživatele a druhá simulací jednoho stochastického uživatele. K oběma těmto sadám byly vygenerovány validační datové sady se stejnými parametry po dobu jednoho simulačního měsíce. Datový modul byl pak nakonfigurován s velikostí dávky (*batch_size*) 256 a délkou sekvencí (*sequence_len*) 1.

Trénování bylo nakonfigurováno s velikostí učícího kroku (*learning_rate*) 0.001 a zachováváním vnitřního stavu modelu v průběhu trénování (*keep_hidden_state: true*). Délka sekvence 1 a zachování vnitřního stavu odpovídá použití modelů v multiagentním systému, kdy model má k dispozici aktuální stav prostředí a předešlé stavy si pamatuje ve svém vnitřním stavu. Po opakovaných experimentech s různými parametry modelu (hyperparametry) jsem pro každý typ neuronové sítě zvolil parametry v tabulce 8.1. Tyto parametry byly použity pro model každého zařízení, tedy pro každé zařízení byl natrénován model s těmito parametry.

Model	Počet skrytých vrstev	Velikost skrytých vrstev
FC	10	512
RNN	3	256
LSTM	3	256
CfC	–	128 (celkový počet neuronů)

Tabulka 8.1: Parametry modelů zařízení

8.1.2 Výsledky

Během simulace je každé zařízení po většinu času vypnuté. To dle mého názoru odpovídá i běžnému provozu reálných zařízení, ovšem to vede k nevyvážené datové sadě. Neuronová síť by tak mohla predikovat, že zařízení je vždy vypnuté, a dosáhnout velké přesnosti (Accuracy) a proto není dobré tuto metriku použít pro vyhodnocení. Hammingova vzdálenost (Hamming) ukazuje množství správných predikcí, a proto, z téhož důvodu jako Accuracy, není vhodnou metrikou pro vyhodnocení. Metriky Precision a Recall jsou zajímavé z pohledu množství nechtěných zapnutí zařízení a jakési míry automatizace, kde nízká hodnota Precision by indikovala velké množství nechtěných zapnutí a vysoká hodnota Recall by indikovala vysokou míru automatizace. Obě tyto metriky by měly být vyváženy a shrnuty metrikou F1Score. Hlavními metrikami pro vyhodnocení jsou tedy metriky F1Score a AUROC, které lépe vystihují správnost predikcí i při nevyvážené datové sadě. Hodnoty metrik počítané v průběhu validace budou přirozeně vyšší než ty v průběhu trénování, jelikož v průběhu trénování se síť teprve učí. Metriky uváděné v následujících tabulkách jsou vypočítány pro celkový predikovaný stav, který se skládá ze stavů všech zařízení.

Deterministický uživatel

Cílem experimentu s datovou sadou deterministického uživatele bylo ověřit, zda budou mít alespoň některé modely dobré výsledky v dobře predikovatelném prostředí a naučí se predikovat pravidelné akce. V tabulce 8.2 jsou uvedeny výsledné metriky počítané v průběhu učení. V tabulce 8.3 jsou uvedeny výsledné metriky validace, kdy se model již neučí.

Model	Accuracy	Precision	Recall	F1Score	AUROC	Hamming
FC	0.9648	0.1268	0.0684	0.0888	0.4361	0.0352
RNN	0.9875	0.8664	0.4975	0.7034	0.5867	0.0125
LSTM	0.9928	0.9572	0.7453	0.8381	0.6081	0.0072
CfC	0.9941	0.9713	0.7873	0.8697	0.6185	0.0059

Tabulka 8.2: Výsledné metriky trénování s daty deterministického uživatele

Model	Accuracy	Precision	Recall	F1Score	AUROC	Hamming
FC	0.928	0.1303	0.3278	0.1865	0.4823	0.072
RNN	0.9961	0.9398	0.9032	0.9211	0.6141	0.0039
LSTM	0.9963	0.8911	0.9699	0.9289	0.6186	0.0037
CfC	0.9979	0.9774	0.9399	0.9583	0.6228	0.0021

Tabulka 8.3: Výsledné metriky validace s daty deterministického uživatele

Z výsledků je patrné, že sítě RNN, LSTM a CfC s velkou úspěšností predikují stavy zařízení a je tak ověřeno, že modely jsou schopné učení a umí vystihnout alespoň pravidelné akce. Nejlepším modelem je podle metrik síť CfC s nejvyššími hodnotami Precision, F1Score a AUROC a nejnižší hodnotou Hammingovy vzdálenosti. Zároveň se ukazuje, že FC síť, i přes vysokou hodnotu Accuracy a relativně nízkou hodnotu Hammingovy vzdálenosti, má špatné výsledky a nedokáže vystihnout pravidelné akce. Tento výsledek se shoduje s očekáváním experimentu.

Stochastický uživatel

Cílem experimentu s datovou sadou stochastického uživatele bylo vybrat a ověřit, zda alespoň některý z modelů má dobré výsledky i ve složitém prostředí a schopnost učení multiagentního systému při výběru některého z modelů. V tabulce 8.4 jsou uvedeny výsledné metriky počítané v průběhu učení. V tabulce 8.5 jsou uvedeny výsledné metriky validace, kdy se model již neučí.

Model	Accuracy	Precision	Recall	F1Score	AUROC	Hamming
FC	0.9582	0.0844	0.0676	0.0751	0.4352	0.0418
RNN	0.9869	0.853	0.6654	0.7476	0.651	0.0131
LSTM	0.9906	0.917	0.7466	0.8231	0.6791	0.0094
CfC	0.9906	0.9166	0.7466	0.8229	0.6874	0.0094

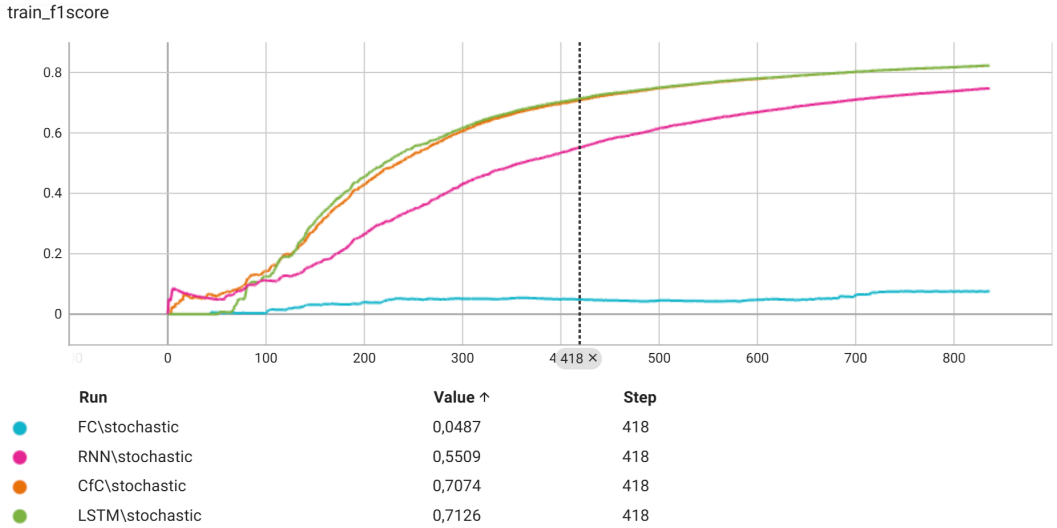
Tabulka 8.4: Výsledné metriky trénování s daty stochastického uživatele

Model	Accuracy	Precision	Recall	F1Score	AUROC	Hamming
FC	0.9323	0.0479	0.0896	0.0624	0.44	0.0677
RNN	0.9951	0.9369	0.8818	0.9085	0.6804	0.0049
LSTM	0.9958	0.9307	0.9135	0.922	0.7198	0.0042
CfC	0.9957	0.9281	0.912	0.92	0.7224	0.0043

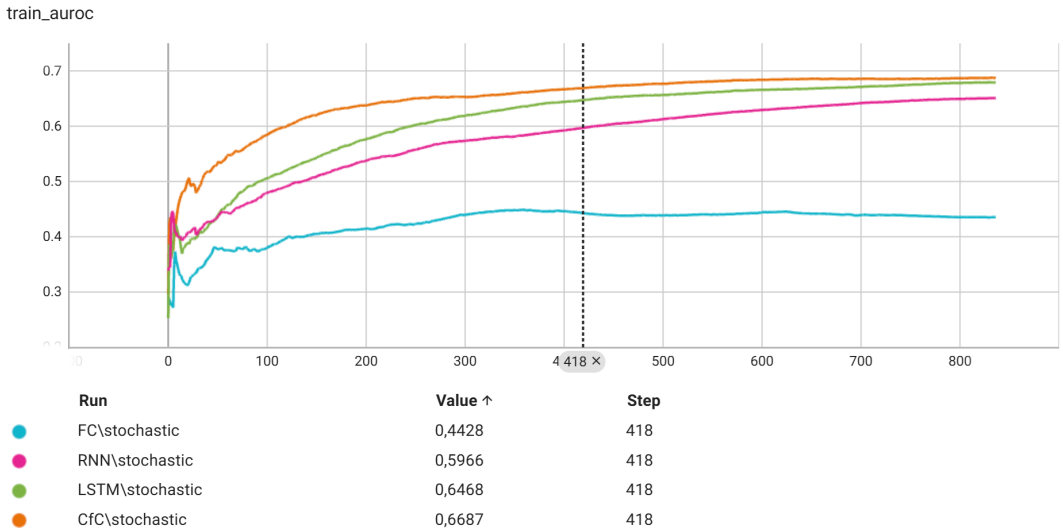
Tabulka 8.5: Výsledné metriky validace s daty stochastického uživatele

Výsledky očekávatelně ukazují, že jednoduchá FC síť opět selhává. Dvou nejlepších a velmi podobných výsledků dosahují sítě CfC a LSTM, které velmi dobře vystihují stochastické prostředí s hodnotami F1Score kolem 0.92 a AUROC kolem 0.72, a lze tak uvažovat o jejich použití v multiagentním systému. RNN síť dosahuje horších výsledků než CfC a LSTM sítě, ale její výsledky také nejsou špatné.

Jelikož se multiagentní systém učí za pochodu (online učení), je z hlediska reálného použití důležité, jak dlouho by se systém musel učit, než by ho bylo možné prohlásit za dostatečně naučený v porovnání s finálními metrikami po dvou letech simulovaných dat. Na grafech 8.1 a 8.2 z nástroje TensorBoard jsou znázorněny průběhy hodnot metrik F1Score a AUROC během trénování. Horizontální osa zobrazuje počet trénovacích kroků/-dávek (batch) a vertikální osa hodnotu dané metriky. Vertikální přerušovaná čára v polovině grafu označuje hodnoty po jednom roce učení a tyto hodnoty jsou vypsané v legendě pod grafem. Barevná křivka je průběhy hodnoty pro daný model. Důležité jsou průběhy pro CfC a LSTM sítě, ostatní modely jsou v grafu jen pro jejich srovnání. Nutno podotknout, že pozdější hodnoty metrik jsou zkresleny špatnými výsledky na začátku trénování.



Obrázek 8.1: Průběh hodnoty F1Score během trénování



Obrázek 8.2: Průběh hodnoty AUROC během trénování

Z průběhů grafů lze sledovat postupné učení, převážně v prvním roce, nejvíce však v prvním půlroce. Osobně bych modely CfC a LSTM prohlásil za dostatečně naučené a použitelné pro predikci po třech čtvrtletích až jednom roce.

8.1.3 Zhodnocení učení

Z provedených experimentů vyplývá, že nejlepších výsledků dosahují CfC a LSTM sítě, a to nejen v deterministickém, ale i stochastickém prostředí. Obě tyto sítě dosahují vysokých hodnot metrik F1Score a AUROC, což naznačuje, že jsou schopny se naučit i z nevyvážených dat.

Průběh metrik během trénování ukazuje, že nejvíce se sítě učí v průběhu prvního roku. Po této době od přidání konkrétního zařízení do multiagentního systému by tak mohly být povoleny jeho akce s vhodně nastavenými prahovými hodnotami.

Další důležitou schopností modelu je rychlost adaptace na změnu chování uživatele. Navazující práce by tak mohla zahrnovat simulaci uživatelů, kteří po určité době změní své chování, a sledovat rychlost adaptace na náhlé anebo postupné změny.

8.2 Běh systému

Cílem této sekce je ověřit funkčnost multiagentního systému jako celku jeho zapojením do implementované simulace a možnost jeho integrace pomocí demonstračního programu `MAS-Simulation/InterfaceTest.py`. Jelikož schopnost učení byla ověřena v předchozí sekci, tato sekce se zaměřuje na používání a běh systému. Funkčnost systému je ověřena úspěšným spuštěním a sledováním korektních výpisů systému při jeho běhu.

Po spuštění XMPP serveru a simulace se zapojeným multiagentním systémem se systém úspěšně spustí v podprocesu a spustí TCP a hlavního agenta. Během inicializace objektů simulace je pro každého simulovaného uživatele úspěšně přidán jeho uživatelský agent a pro každé zařízení je úspěšně přidán predikční agent, který podle konfigurace načte uložený natrénovaný model. Následně jsou úspěšně nastaveny uživatelské filtry každému uživatelskému agentovi pro každé zařízení podle konfigurace. V průběhu simulace systém predikuje stavy zařízení a zasílá akce vygenerované interpretováním jednotlivých predikcí, které jsou provedeny s konkrétním simulačním modelem zařízení. Po skončení běhu simulace se multiagentní systém úspěšně ukončí a podle konfigurace uloží natrénované modely. Multiagentní systém průběžně vypisoval očekávané výstupy.

Funkčnost systému a jeho možnost integrace skrze implementované rozhraní byla ověřena demonstračním programem `MAS-Simulation/InterfaceTest.py` se spuštěným XMPP serverem. Rozhraní úspěšně spustí multiagentní systém v podprocesu a počká na spuštění hlavního agenta. Po spuštění systému lze úspěšně přidávat uživatelské a predikční agenty. Dále lze uživatelským agentům úspěšně nastavit uživatelský filtr. Stav prostředí se správně zasílá a propaguje k predikčním agentům. Po zaslání zprávy pro predikci na vyžádání vygenerují predikční agenti predikce, které jsou zaslány uživatelským agentům. Uživatelští agenty korektně filtrují a interpretují predikce a případně zasílají vygenerované akce. Zaslané akce jsou pak úspěšně získány skrze rozhraní. Nakonec je multiagentní systém korektně ukončen. Multiagentní systém průběžně vypisoval očekávané výstupy.

8.3 Přechod k reálnému nasazení

Multiagentní systém lze integrovat do existujícího řídícího systému pomocí implementovaného rozhraní a po zhruba roce učení ho lze použít pro predikci stavů chytrých spínačů. Mimo jiné používá multiagentní systém k predikci informaci o lokaci uživatele, které nemusí být k dispozici. Tuto informaci však multiagentní systém nutně nepotřebuje a lze ho použít

i bez této informace, ale může tak dojít ke zhoršení učení a přesnosti predikce. Způsoby získání uživatelské lokace jsou popsány v [1].

Problémem realizovaného multiagentního systému je, že neřeší konflikty mezi akcemi jednotlivých uživatelských agentů. Tato úloha tedy připadá na implementaci integrace systému do existujícího řídicího systému a lze ji řešit např. prioritami uživatelů. Navazující práce by tento problém mohla vyřešit pomocí hlavního agenta, který by konflikty řešil.

Dalším problémem je doba potřebná k učení, kvůli které nemusí být multiagentní systém zcela praktický. Potřebná doba učení jednoho roku je navíc stanovena při neměnném chování uživatele z pohledu interakce s daným zařízením. Implementaci predikčního modelu v predikčních agentech lze rozšířit i o jiné neuronové sítě nebo zcela jiné způsoby učení, které by mohly vést k rychlejšímu procesu učení, a je námětem k navazující práci.

Navazující práce by se také mohla pokusit o rozšíření systému o další podsystémy, jako např. chytré vytápění, a o zařízení se spojitým ovládáním, jako např. termostat. Systém by pak mohl např. postupně vytápět místnost s blížícím se uživatelem.

Kapitola 9

Závěr

Cílem této práce bylo vytvořit multiagentní systém, který se učí predikovat stavy chytrých spínačů v rámci Smart Home a na základě uživatelského nastavení generovat akce za účelem automatizace. Tento cíl se podařilo splnit. Byla navržena a implementována hierarchická architektura multiagentního systému, který k predikci využívá neuronové sítě a jednoduchou logiku uživatelských filtrů a je schopný se učit predikovat stavy chytrých spínačů.

V rámci práce bylo vytvořeno simulační prostředí s modely chytrých spínačů a dvěma modely uživatelů, které poskytuje data pro trénování neuronových sítí a slouží jako prostředí pro ověření funkčnosti multiagentního systému. Byli navrženi a implementováni jednotliví agenti, jmenovitě hlavní agent pro koordinaci a přeposílání zpráv, predikční agenti pro predikci stavů jednotlivých spínačů a uživatelští agenti pro filtraci a interpretaci predikcí jako akce. Byly otestovány vybrané architektury neuronových sítí (včetně LSTM a CfC) a ověřena jejich schopnost učení na simulačně vygenerovaných datech. Funkčnost multiagentního systému jako celku byla ověřena v simulačním prostředí a programem demonstrujícím jeho možnost integrace do existujícího řídícího systému.

Při ověření schopnosti učení systému na simulačních datech zaznamenaných po dobu dvou simulačních let dosáhly LSTM a CfC neuronové sítě výsledných hodnot kolem 0.92 pro metriku F1Score a kolem 0.72 pro metriku AUROC při predikci stavu zařízení s intervalem 5 minut. Tyto metriky ukazují vysokou přesnost spínání a schopnost rozlišení stavů zařízení. Neuronové sítě lze prohlásit za dostatečně naučené po třech čtvrtletích až jednom roce učení.

Navazující práce by mohla multiagentní systém rozšířit o další typy zařízení a subsystémy, jako například termostaty a subsystém vytápění. Informace o stavu prostředí by mohla být rozšířena o další senzorická data, která by mohla vést k přesnějším predikcím. Dalším možným námětem je experimentování s jinými architekturami neuronových sítí anebo zcela jinými způsoby strojového učení, jako například rozhodovací stromy. Užitečným rozšířením systému by bylo řešení konfliktních akcí mezi uživatelskými agenty v situacích, kdy mají uživatelé rozdílné preference pro stejné zařízení. V neposlední řadě by mohl být systém integrován do existujícího řídícího systému a ověřen v reálném nasazení.

Literatura

[1] Ahvar, E.; Daneshgar Moghaddam, N.; Ortiz, A. M.; Lee, G. M. a Crespi, N. On analyzing user location discovery methods in smart homes: A taxonomy and survey. *Journal of Network and Computer Applications*, 2016, sv. 76, s. 75–86. ISSN 1084-8045. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1084804516302181.

[2] Budu, E. *Online Learning vs. Offline Learning.* March 2025. Dostupné z: https://www.baeldung.com/cs/online-vs-offline-learning.

[3] Chakraborty, A.; Islam, M.; Shahriyar, F.; Islam, S.; Zaman, H. U. et al. Smart Home System: A Comprehensive Review. *Journal of Electrical and Computer Engineering*, 2023, sv. 2023, s. 1–30.

[4] Cook, D. J.; Youngblood, M.; III, E. O. H.; Gopalratnam, K.; Rao, S. et al. MavHome: An Agent-Based Smart Home. In: IEEE Computer Society. *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom 2003)* online. Fort Worth, TX, USA: IEEE, Březen 2003, s. 521–524. ISBN 0-7695-1893-1. Dostupné z: https://doi.org/10.1109/PERCOM.2003.1192783. [cit. 2025-04-24].

[5] DataCamp Team. *Time Series Forecasting Tutorial* DataCamp Tutorial. 2025. Dostupné z: https://www.datacamp.com/tutorial/tutorial-time-series-forecasting. [cit. 2025-04-03].

[6] Foundation for Intelligent Physical Agents. *FIPA Communicative Act Library Specification.* FIPA, 2002. Dostupné z: http://www.fipa.org/repository/aclspecs.html. [cit. 2025-04-03].

[7] Hasani, R.; Lechner, M.; Amini, A.; Liebenwein, L.; Ray, A. et al. Closed-form continuous-time neural networks. *Nature Machine Intelligence*, nov 2022, sv. 4, č. 11, s. 992–1003. Dostupné z: https://www.nature.com/articles/s42256-022-00556-7.

[8] Hasani, R. M.; Lechner, M.; Amini, A.; Rus, D. a Grosu, R. Liquid Time-constant Networks. *CoRR*, 2020, abs/2006.04439. Dostupné z: https://doi.org/10.48550/arXiv.2006.04439.

[9] Home Assistant Community. *All The Silly Home Releases* online. 2022. Dostupné z: https://community.home-assistant.io/t/all-the-silly-home-releases/447305. [cit. 2025-04-27].

[10] Janoušek, V. *Tekuté neuronové sítě. Materiály k přednáškám předmětu Softcomputing na FIT VUT v Brně.*

[11] Kriesel, D. *A Brief Introduction to Neural Networks*. 2007. Dostupné z: http://www.dkriesel.com.

[12] Lai, C. *The Silly Home: Containerized AI-Based Smart Home Automation* online. Dostupné z: https://github.com/lcmchris/thesillyhome-container. [cit. 2025-04-25].

[13] Lechner, M. *Neural Circuit Policies*. 2020. Dostupné z: https://github.com/mlech26l/ncps.

[14] Lightning AI. *PyTorch Lightning*. 2025. Dostupné z: https://lightning.ai/.

[15] Lightning AI. *TorchMetrics Documentation*. 2025. Dostupné z: https://lightning.ai/docs/torchmetrics/stable/.

[16] Palanca, J. *Welcome to SPADE's documentation! — SPADE 3.3.0 documentation* online. Dostupné z: https://spade-mas.readthedocs.io/. [cit. 2024-04-06].

[17] PyTorch Contributors. *PyTorch: An Open Source Machine Learning Framework* online. Dostupné z: https://pytorch.org/. [cit. 2024-05-03].

[18] Russell, S. a Norvig, P. *Artificial Intelligence: A Modern Approach*. 3. vyd. Prentice Hall, 2016. ISBN 978-0-13-604259-4.

[19] Russell, S. a Norvig, P. *Artificial Intelligence: A Modern Approach: Global Edition*. 4. vyd. Pearson Education Limited, 2021. ISBN 978-1-292-40113-3.

[20] Sahin, S. *Multi-Agent AI Systems: Foundational Concepts and Architectures*. 2023. Dostupné z: https://medium.com/@sahin.samia/multi-agent-ai-systems-foundational-concepts-and-architectures-ece9f8859302.

[21] Team SimPy. *SimPy - Discrete event simulation for Python* online. Dostupné z: https://simpy.readthedocs.io/en/latest/. [cit. 2024-04-09].

[22] TensorFlow Team. *TensorFlow: An End-to-End Open Source Machine Learning Platform*. 2025. Dostupné z: https://www.tensorflow.org/.

[23] Tharwat, A. Classification Assessment Methods. *Applied Computing and Informatics*, 2021, sv. 17, č. 1, s. 168–192. Dostupné z: https://doi.org/10.1016/j.aci.2018.08.003.

[24] Wooldridge, M. *An introduction to multiagent systems*. 2. vyd. John Wiley & Sons, 2009. ISBN 978-0-470-51946-2.

[25] Zbořil, F. *Materiály k přednáškám předmětu Agentní a multiagentní systémy na FIT VUT v Brně.*

[26] Zbořil, F. *Rekurentní neuronové sítě pro zpracování časových řad. Materiály k přednáškám předmětu Softcomputing na FIT VUT v Brně.*