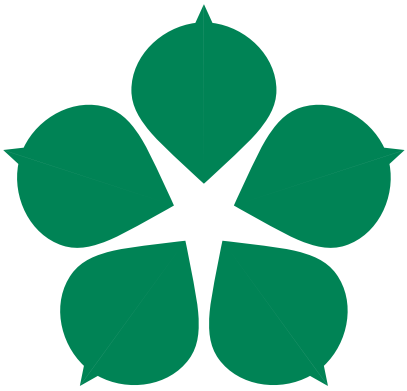
Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta



Optimalizační nástroj pro modely s hlubokým učením

Bakalářská práce

Václav Filip

Vedoucí práce: Ing. Jiří Jelínek, CSc.

České Budějovice 2021

Bibliografické údaje

Filip, V. 2021: Optimalizační nástroj pro modely s hlubokým učením. [Optimization tool for deep learning models. Bc. Thesis, in Czech.] – 37 p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic

Anotace

The goal of this thesis is to create a tool capable of optimizing parameters and structure of deep learning models. There are various ways to create a model like changing types and number of layers, various numerical parameters of the layers etc. These parameters greatly influence the quality of the model. Newly created tool is to offer functionalities not offered by any existing solution - these new functionalities are based on comparison of existing solutions. New solution is created in python.

Prohlašuji, že jsem autorem této kvalifikační práce a že jsem ji vypracoval pouze s použitím pramenů a literatury uvedených v seznamu použitých zdrojů.

V ČESKÝCH BUDĚJOVICÍCH **dne** 28.11.2021 **Podpis** Filip

Poděkování

Děkuji vedoucímu mé bakalářské práce panu Ing. Jiřímu Jelínkovi, CSc. za jeho rady, doporučení a ochotu při její tvorbě, včetně jeho odborného vedení a trpělivosti.

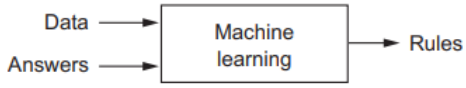
Obsah

1. Úvod

1.1 Popis

Tato práce si klade za cíl vytvoření optimalizačního nástroje pro modely neuronových sítí s hlubokým učením v prostředí funkcionálního API knihovny Keras.

V první řadě je nutné říci, co je vlastně strojové učení. Strojové učení je proces, při kterém je naším cílem naučit výpočetní systém pracovat s určitým druhem dat. Představa je následující: modelu je poskytnuto značné množství relevantních dat, spolu s kterými může být poskytnut i očekávaný výstup, ale také nemusí pokud chceme, aby model data zpracoval samostatně. Cílem je, aby model nalezl v těchto datech souvislosti a zároveň si nastavil váhy jednotlivých vrstev tak, aby následně dokázal poskytnout správný výstup i na nově poskytnutá data vztahující se k problému, ale se kterými se dosud nesetkal. Tento princip je ukázán na obrázku 1.1. Příkladem, při kterém jsou modelu poskytnuta data společně s očekávaným výstupem, může být například klasifikace obrázků - vstupním datasetem jsou obrázky, které již byly nějakým způsobem klasifikovány - například lidmi, přičemž model se v průběhu učícího procesu naučí pravidla pro klasifikaci těchto obrázků a následně bude schopen správně klasifikovat podobné obrázky s určitou přesností.



Obrázek 1.1: Ilustrativní obrázek zobrazující princip strojového učení - dostupné z [1]

Deep learning je součástí strojového učení. V podstatě jde o to, že mezi vstupní a výstupní vrstvou se nachází množství po sobě jdoucích vrstev. Množství těchto vrstev nám definuje hloubku modelu. Hlavním rozdílem od ostatních přístupů strojového učení je právě množství vrstev - deep learning jich může obsahovat desítky až stovky a všechny se podílí na funkcionalitě modelu, přičemž ostatní přístupy typicky trénují pouze jednu až dvě vrstvy reprezentující data.

1.2 Cíle práce

Účelem práce je vytvoření nástroje schopného modifikovat modely a testovat různé varianty nastavení s cílem najít optimální model pro daná data. Představme si, že chceme vytvořit vlastní deep learning model - pak máme mnoho možností, jakým způsobem tento model sestavit s využitím různých parametrů (například typy použitých vrstev, jejich počet, pořadí apod.), přičemž podoba modelu má zásadní vliv na jeho kvalitu. Existuje jen několik málo aplikací, které tuto problematiku částečně řeší a umožňují automatické testování různých variant nastavení parametrů, komplexní nástroj tohoto typu umožňující optimalizaci struktury by byl přínosem i pro projekty UAI.

Práce si klade za cíl dosáhnout následujících bodů:

1. Odborná rešerše existujících řešení, popis jejich funkcionality, klady a zápory.
2. Specifikace funkcionality a případů užití zamýšlené aplikace na základě výstupu bodu 1. Výběr technických prostředků a jejich stručný popis.
3. Definice parametrů pro optimalizaci, výběr optimalizační metody na základě provedené rešerše a porovnání.
4. Návrh aplikace dle zásad návrhu SW produktu s využitím příslušných diagramů. Vytvoření aplikace v jazyce Python na základě navržené funkcionality.
5. Otestování požadované funkcionality aplikace - využití vybraných testovacích datasetů a porovnání úspěšnosti optimalizovaného modelu s ostatními řešeními.

1.3 Keras

V této problematice je využíván Keras. Zde je vhodné se krátce zmínit o tom, k čemu vlastně slouží. Keras je knihovna jazyka python, která umožňuje manipulaci s modely strojového učení - jejich definici, tvorbu, natrénování a také nasazení na dataset. Keras nám umožňuje definovat modely dvěma způsoby - definovat takzvané sekvenční modely, nebo využít k tvorbě modelů takzvané funkcionální API.

Sekvenční model

První možností je takzvaný sekvenční model. Tento způsob tvorby modelu je vhodný, pokud mají jít vrstvy po sobě a každá z nich má mít pouze 1 vstupní a výstupní tensor. Nepoužívá se, pokud chceme, aby náš model obsahoval vrstvy s více vstupními či výstupními tensory, nebo několikanásobné větvení struktury a propojení vrstev. Sekvenční modely lze vytvářet pomocí konstruktoru třídy „keras.Sequential“, způsobem ukázaným na obrázku 1.2, kdy je struktura modelu předávána jako parametr, nebo lze také postupovat samotným vytvořením modelu a následným přidáváním vrstev do struktury pomocí metody „model.add“. Tento způsob je ukázán na obrázku 1.3.

```
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu", name="layer1"),
        layers.Dense(3, activation="relu", name="layer2"),
        layers.Dense(4, name="layer3"),
    ]
)
```

Obrázek 1.2: Ilustrativní obrázek tvorby sekvenčního modelu - převzato z [3]

```
model = keras.Sequential(name="my_sequential")
model.add(layers.Dense(2, activation="relu", name="layer1"))
model.add(layers.Dense(3, activation="relu", name="layer2"))
model.add(layers.Dense(4, name="layer3"))
```

Obrázek 1.3: Ilustrativní obrázek tvorby sekvenčního modelu s využitím metody add - převzato z [3]

Funkcionální API

Druhou možností je takzvané funkcionální API. Tento způsob nám umožňuje vytvářet modely, které jsou značně flexibilnější, než-li modely sekvenční. Myšlenkou tohoto přístupu je, že deep learning modely jsou koncipovány jako acyklické orientované grafy a mohou být tímto způsobem také vytvářeny. Z tohoto pohledu je proto logické, že prvním krokem je vytvoření vstupního uzlu takovéhoto grafu a následné přidávání uzlů, které představují jednotlivé neurony v daných vrstvách. Tvorba těchto uzlů je ukázána na obrázku 1.4. Řádek „x = dense(input)“ nám definuje směr, kterým bude graf postupovat, tedy od vrstvy input vede pomyslná šipka k vrstvě dense našeho orientovaného grafu. Tento přístup nám umožňuje definovat ku příkladu modely, které obsahují několik větví, různá propojení mezi vrstvami a také využít existující model jako vrstvu a vložit jej do modelu nového.

```
dense = layers.Dense(64, activation="relu")
x = dense(inputs)
```

Obrázek 1.4: Ilustrativní obrázek tvorby vrstvy ve funkcionálním API - převzato z [3]

2. Existující řešení

V této sekci uvádím řešení, která již existují a podobnou problematiku nějakým způsobem řeší. Optimalizace v těchto řešeních probíhá principem testování různých možností - optimalizované hyperparametry jsou v nějakém nastavení zvoleny, takovýto model je otestován a stejným způsobem s jiným nastavením je postupováno i v dalších iteracích testu.

2.1 Sekvenční modely

Mezi řešení podporující sekvenční modely patří Talos, Keras-Tuner, Ray-Tune a Hyperas. Talos a Keras-Tuner podporují navíc také funkcionální API a jsou popsány v části Funkcionální API, proto se jimi v této sekci nebudu zabývat.

2.1.1 Ray-Tune

Prvním příkladem nástroje umožňujícím optimalizaci pouze sekvenčních modelů je Ray-Tune. V případě využití Ray-Tune uživatel nejprve vytvoří strukturu modelu, pro kterou chce optimalizaci provést a takzvaný „config“ (python dictionary) obsahující možná nastavení parametrů ve formátu „jméno“:„parametr“ - parametrem může být konkrétní hodnota, nebo následující metody: „tune.uniform()“ - tato metoda vrací číslo formátu „float“ v rovnoměrném rozložení (pro toto rozdělení existují obdobné metody pro čísla formátu „int“) z rozsahu od - do specifikováno coby parametry této funkce (funkce má několik obdob, například „tune.quniform()“ obsahující navíc parametr zaokrouhlení), dále můžeme využít funkci „tune.randn()“ (případně „tune.qrandn()“ při zaokrouhlení) pro „float“ z normálního rozdělení. Další možností je metoda „tune.choice()“, jejímž parametrem je list možných hodnot, z nichž chceme jednu vybrat. Taktéž můžeme pomocí metody „tune.sample_from“ vytvořit funkci, která bude vycházet z již definovaného parametru ve slovníku (například využití již existujícího číselného rozsahu s tím, že výsledek bude vynásoben nějakým koeficientem).

Příklad takovéhoto configu je uveden na obrázku 2.1. Na tomto obrázku je config definován přímo při volání metody „tune.run()“ (je možné jej ale definovat mimo a pouze předávat referenci), dalším parametrem této metody je zde „trainable“, což je funkce, ve které se parametry z configu dosazují na definovaná místa. U metody „tune.run()“ lze využít taktéž tzv. „schedulery“ - jedná se o algoritmy, které se snaží upravovat testování konkrétních konfigurací na základě metriky,

například dát delší časový úsek na otestování configu, který má větší šanci uspět.

```
tune.run(
    trainable,
    config={
        "param1": tune.choice([True, False]),
        "bar": tune.uniform(0, 10),
        "alpha": tune.sample_from(lambda _: np.random.uniform(100) ** 2),
        "const": "hello"  # It is also ok to specify constant values.
    })
```

Obrázek 2.1: Ilustrativní obrázek zobrazující příklad slovníku - převzato z [2]

Takovýto config je vygenerován pro každé kolo optimalizace. Po skončení je vrácen config s nastavením odpovídajícím nejlépe dosaženému výsledku v průběhu testování.

2.1.2 Hyperas

Dalším řešením umožňující optimalizaci pouze sekvenčních modelů je Hyperas. Toto řešení je wrapper nad knihovnou hyperopt umožňující její využití pro Keras bez nutnosti používání specifických syntaxí knihovny. Uživatel musí definovat dvě funkce - a to první, která vrací Keras model a druhou, která vrací trénovací a testovací data potřebná k učení modelu.

Hyperparametry uživatel označuje přímo v definici modelu pomocí dvojitých složených závorek (například označení {{uniform(0,1)}}, které je v praxi využito na obrázku 2.2, znamená využití rovnoměrného rozdělení v intervalu 0 až 1). Dalšími nabízenými možnostmi jsou funkce „choice()“ - vrací prvek z listu či tuple, „randint()“ - vrací integer mezi 0 a parametrem zadanou hodnotou, „uniform()“ - s parametry od a do vrací číslo v tomto rozsahu s rovnoměrným rozdělením, „normal()“ - s parametry střední hodnota a rozptyl. Ještě stojí za zmínku možnost využití několika funkcí, které jsou definovány určitým specifickým způsobem, jejichž popis zde ale nebudu pro přehlednost uvádět, přičemž příkladem za všechny bude funkce s parametry „quniform(label, low, high, q)“, která vrací hodnotu podle následující definice: „round(uniform(low, high) / q) * q“.

```
from hyperas.distributions import uniform

def create_model(x_train, y_train, x_test, y_test):
    model = Sequential()
    model.add(Dense(512, input_shape=(784,)))
    model.add(Activation('relu'))
    model.add(Dropout({{uniform(0, 1)}}))
    model.add(Dense(512))
    model.add(Activation('relu'))
    model.add(Dropout({{uniform(0, 1)}}))
    model.add(Dense(10))
    model.add(Activation('softmax'))

    # ... model fitting

    score = model.evaluate(x_test, y_test, verbose=0)
    accuracy = score[1]
    return {'loss': -accuracy, 'status': STATUS_OK, 'model': model}
```

Obrázek 2.2: Ilustrativní obrázek zobrazující definici parametrů v řešení Hyperas - převzato z [6]

Po definování příslušných funkcí můžeme s použitím metody „optimize()“ (které tyto funkce předáváme jako parametr spolu s počtem pokusů) přejít k samotné optimalizaci, kdy se za využití definovaného algortimu snažíme minimalizovat zadanou metriku, podle které má být model optimalizován (například accuracy - tuto bychom museli definovat coby zápornou hodnotu, protože hyperopt vždy danou metriku minimalizuje). Na výběr máme z náhodného výběru a „Tree of Parzen Estimators (TPE)“ (způsob výběrů parametrů se po několika iteracích řídí heuristicky s ohledem na historicky dosažených výsledků s cílem dosáhnout optimálního návrhu). Výstupem této metody je model s nejlepším dosaženým výsledkem na základě námi definované metriky.

2.2 Funkcionální API

2.2.1 Talos

Prvním z existujících řešení, které podporuje tvorbu modelů pomocí funkcionálního API je Talos.

Talost funguje na principu uživatelem definovaného slovníku - uživatel definuje před samotnou optimalizací pythonovskou strukturu typu dictionary - slovník, která obsahuje dvojice „hyperparametr“ : „prvky“ - tyto prvky mohou být definovány jako list, který obsahuje výčet hodnot, jichž může parametr nabývat, nebo jako rozsah hodnot ve formátu (minimum, maximum, krok). Příklad takového slovníku je uveden na obrázku 2.3.

```
p = {'lr': (0.1, 10, 10),
     'first_neuron':[4, 8, 16, 32, 64, 128],
     'batch_size': [2, 3, 4],
     'epochs': [200],
     'dropout': (0, 0.40, 10),
     'optimizer': [Adam, Nadam],
     'loss': ['categorical_crossentropy'],
     'last_activation': ['softmax'],
     'weight_regulizer': [None]}
```

Obrázek 2.3: Ilustrativní obrázek zobrazující příklad slovníku potřebného pro Talos optimalizaci - převzato z [4]

Po nadefinování slovníku musí uživatel specifikovat strukturu modelu (pořadí vrstev apod.) a uvést klíčová slova params(„konkrétní hyperparametr“) do příslušných částí, kam má být daná hodnota dosazena - toto je ukázáno na obrázku 2.4.

```
def iris_model(x_train, y_train, x_val, y_val, params):

    model = Sequential()
    model.add(Dense(params['first_neuron'],
                    input_dim=x_train.shape[1],
                    activation='relu'))

    model.add(Dropout(params['dropout']))
    model.add(Dense(y_train.shape[1],
                    activation=params['last_activation']))

    model.compile(optimizer=params['optimizer'](lr=lr_normalizer(params['lr'], params['optimizer'])),
                  loss=params['loss'],
                  metrics=['acc'])

    out = model.fit(x_train, y_train,
                    batch_size=params['batch_size'],
                    epochs=params['epochs'],
                    verbose=0,
                    validation_data=[x_val, y_val])

    return out, model
```

Obrázek 2.4: Ilustrativní obrázek zobrazující strukturu modelu pro Talos - převzato z [4]

Následně zavoláme funkci „talos.scan()“, která provede na konkrétním modelu samotnou optimalizaci - při ní Talos testuje množství permutací tvořených z parametrů ve slovníku zadaným způsobem (parametrem „fraction“ při volání funkce „scan“ můžeme omezit množství testovaných permutací). Podle dokumentace Talos podporuje několik optimalizačních strategií, mezi nimiž například „grid search“, „random search“ a „probabilistic reduction“.

Výsledkem funkce „talos.scan()“ je takzvaný scan objekt, který obsahuje několik užitečných metod, mezi nimiž například „saved_models“ a „saved_weights“, které nám umožňují přístup k vahám a modelům všech testovaných permutací. Neméně důležitá je metoda „talos.Analyze()“, jejímž parametrem je výstup funkce „scan“.

Tato nám umožňuje vyhledávat v otestovaných variantách pomocí několika metod - mezi nimiž například „high()“ s parametrem metriky, podle které chceme vyhledávat - například použití metody „high(„acc“)“ nám vrátí nejvyšší dosaženou přesnost v průběhu testování. Konkrétní modely lze pomocí metody „Evaluate()“, jejímž parametrem je buďto logovací soubor, nebo objekt získaný po volání funkce „scan()“, ohodnotit podle námi zvolené metriky. Na tento konkrétní model (modely) lze volat metodu „Deploy()“, která stav modelu uloží coby zazipovaný soubor. Takto uložený model lze následně nahrát pomocí funkce „Restore()“, jejímž parametrem je zazipovaný soubor obsahující model.

2.2.2 Keras-Tuner

Druhým existujícím řešením podporujícím funkcionální API je Keras-Tuner.

Toto řešení využívá k definování optimalizovaných hyperparametrů třídu „HyperParametrs“, jejíž instanci budu pro jednoduchost v následující části nazývat hp. Příkladem definice hyperparametru typu integer je „hp.Int()“, který definuje jako parametr Integer v určitém rozsahu. Dále můžeme definovat hyperparametry pomocí metod „boolean()“ - nabývající hodnot True či False, metody „Choice()“ - výběr z předem definovaného listu, metody „Fixed()“ - daný parametr bude vždy konstantní, metody „Float()“ - vrací číselnou hodnotu typu float, či metody „Conditional_scope()“ - parametry takto definované se využijí pouze, pokud bude splněna definovaná podmínka. Příklad definování parametrů pomocí metody „Int()“ a „Choice()“ je ukázán na obrázku 2.5.

```
def build_model(hp):
    model = keras.Sequential()
    model.add(layers.Flatten())
    model.add(
        layers.Dense(
            units=hp.Int("units", min_value=32, max_value=512, step=32),
            activation="relu",
        )
    )
    model.add(layers.Dense(10, activation="softmax"))
    model.compile(
        optimizer=keras.optimizers.Adam(
            hp.Choice("learning_rate", values=[1e-2, 1e-3, 1e-4])
        ),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

Obrázek 2.5: Ilustrativní obrázek zobrazující definování parametrů - převzato z [5]

Dalším krokem je vytvoření instance tzv. tuneru, který definuje strategii výběru konkrétních hodnot hyperparametrů v testech. Keras-tuner podporuje například „BayesianOptimization“, „RandomSearch“, „Hyperband“ a pomocí třídy „Tuner“ lze vytvořit tuner vlastní. Příklad vytvoření tuneru typu „RandomSearch“ je uveden na obrázku 2.6. Nutnými parametry jsou funkce vracející model (místo této funkce lze využít třídu „HyperModel“ implementující metodu „build()“, která vrací vytvořený model), metrika optimalizace, množství testů a kolik modelů má být v každém testu vytvořeno.

```
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,
    executions_per_trial=3,
    directory='my_dir',
    project_name='helloworld')
```

Obrázek 2.6: Ilustrativní obrázek zobrazující vytvoření tuneru - převzato z [5]

Následně můžeme získat přehled nastavení pomocí volání metody „search_space_summary()“ a poté přejít k samotnému testování metodou „search()“. V průběhu tohoto prohledávání si tuner ukládá informace ke každému modelu a po skončení můžeme k získání nejlepších otestovaných modelů využít metodu „get_best_models()“ s parametrem „num_models“ určujícím počet modelů.

3. Nové řešení

3.1 Shrnutí existujících řešení

Všechna zmíněná existující řešení umožňují částečnou optimalizaci parametrů, které představují metody zmíněné u konkrétních řešení. Hyperas a Ray-Tune umožňují částečné začlenění různých pravděpodobnostních rozdělení do definice parametrů ve smyslu využití specifických metod, při kterém dojde k vygenerování čísla buď z normálního, nebo rovnoměrného rozdělení - v jiných ohledech nelze s pravděpodobnostními rozděleními pracovat. Talos tuto možnost nenabízí vůbec a Keras-Tuner umožňuje využití rozdělení lineárního a logaritmického ve výběru čísla (float či int). Řešení pracují buď s definicí parametrů externě v nějakém listu s tím, že daný parametr je následně dosazen na označené místo v modelu, nebo jsou parametry přímo označovány v definici modelu voláním konkrétní metody, která daný parametr nějakým způsobem vybírá. Dále je v každém řešení vyžadována definice modelu od uživatele v plném rozsahu - s výjimkou použití for cyklů a jejich parametrizace (například vytvoření 0-n vrstev stejného typu za sebou) při navrhování modelu, dosažený výsledek ale stále odpovídá pouze znásobení určité vrstvy x-krát v řadě.

3.1.1 Nedostatky

Nové řešení si klade za cíl odstranit určité nedostatky, které výše uvedená řešení obsahují. Prvním z nich je fakt, že od uživatele je vyžadována definice modelu sítě v plném rozsahu - lze parametrizovat aspekty vrstev a jejich počty ve smyslu for cyklu, který vytvoří 0-x vrstev stejného typu za sebou, ale jinak musí celý model definovat před samotnou optimalizací uživatel. V novém řešení bude moci uživatel vyjma plné definice modelu využít možnosti generování části modelu aplikací - toto bude založeno na uživatelem definovaném slovníku, pomocí kterého aplikace vytvoří strukturu. V souvislosti s touto problematikou je vhodné podotknout, že existující řešení neumožňují měnit pořadí či typy definovaných vrstev. Dalším nedostatkem je způsob, kterým některá z již existujících řešení pracují s možností využití různých pravděpodobnostních rozdělení. Možnosti jsou omezeny pouze na generování čísel z rovnoměrného a normálního rozdělení s tím, že uživatel musí využít specifickou funkci, která toto podporuje a vždy jde pouze o využití při vygenerování čísla (float či int). V novém řešení bude uživatel volat funkci, jejímž parametrem bude označení pravděpodobnostního rozdělení, které má být využito

(například volání funkce „range(0, 500, „normal“)“ vrací číslo z intervalu 0-500 s využitím normálního rozdělení). Parametr rozdělení je nepovinný, v případě jeho neuvedení bude využito defaultně nastavené rozdělení.

3.2 Funkcionalita

3.2.1 Zamýšlené užití

Aplikace je určena k vytvoření a optimalizaci deep learning modelu, který si uživatel přeje. Samotný nástroj bude navržen v jazyce python za využití prostředí jupyter notebook, které umožňuje práci s kódem v buňkách, modifikaci případných nastavení, spouštění jednotlivých buněk a další práci s výsledky například pomocí grafů. Stručný návod k použití je k dispozici v příloze 1 kapitoly Přílohy. Funkcionalita nového řešení je zamýšlena následovně:

1. Před spuštěním aplikace uživatel vykoná následující akce:
    - Definice datasetu, pro který bude síť optimalizována.
    - Definice modelu v příslušné metodě, případně ostatní definice nutné pro funkcionalitu v případě využití kupříkladu generování struktury.
    - Definice a označení samotných parametrů, které chceme optimalizovat (například výběr z listu).
    - Definice parametrů funkcionality při volbě optimalizační metody (množství vytvářených a testovaných modelů, počet cyklů po které budeme testovat nové modely apod.).
2. Samotné spuštění aplikace:
    - Dochází k tvorbě zadaného množství modelů dle nastavení - modely jsou otestovány na datasetu a ohodnocovány na základě námi definovaného parametru.
    - Následně je aplikována optimalizační metoda (například genetické algoritmy) a nově vytvořené modely jsou opět testovány.
    - Takto aplikace pokračuje po zadaný počet cyklů a po skončení proces vrací model s nejlepšími výsledky.
    - Následně lze s modely získanými procesem dále pracovat, například je uložit.

3. Porovnání s existujícími řešeními:

    - Pokud bude uživatel chtít, může na stejném datasetu porovnat výsledek vybraného modelu s modelem, který odněkud převzal.

3.2.2 Podporované hyperparametry

V aplikaci je možné optimalizovat prakticky jakékoliv parametry, které odpovídají následujícím datovým strukturám:

- **Výčet** - definice různých možností, kterých může konkrétní parametr nabývat (definováno jako struktura list) - funkce bude volána následujícím zápisem: „Choice(list, distribution)“ s tím, že parametry jsou povinně list možností a nepovinně pravděpodobnostní rozdělení - v případě neuvedení se použije defaultní nastavení a výsledkem je vrácení konkrétního prvku z listu.

- **Rozsah** - definován minimální a maximální hodnotou (například v rozsahu 20 - 1000) - funkce bude volána zápisem „numRange(x, y, distribution, a)“ s tím, že parametry jsou povinně od, do a nepovinně pravděpodobnostní rozdělení a zaokrouhlení na určité desetinné místo (použije se v případě float hodnot).

- **Rozsah s krokem** - funkce bude volána zápisem „stepRange(x, y, z, distribution, a)“ s tím, že parametry jsou povinně od, do, krok a nepovinně pravděpodobnostní rozdělení a zaokrouhlení na určité desetinné místo (použije se v případě float hodnot).

- **Integer** - rozsah celočíselných hodnot - zapisován metodou „numRange()“ s tím, že konkrétní implementaci obsahuje vlastní metoda „intRange()“ daného pravděpodobnostního rozdělení.

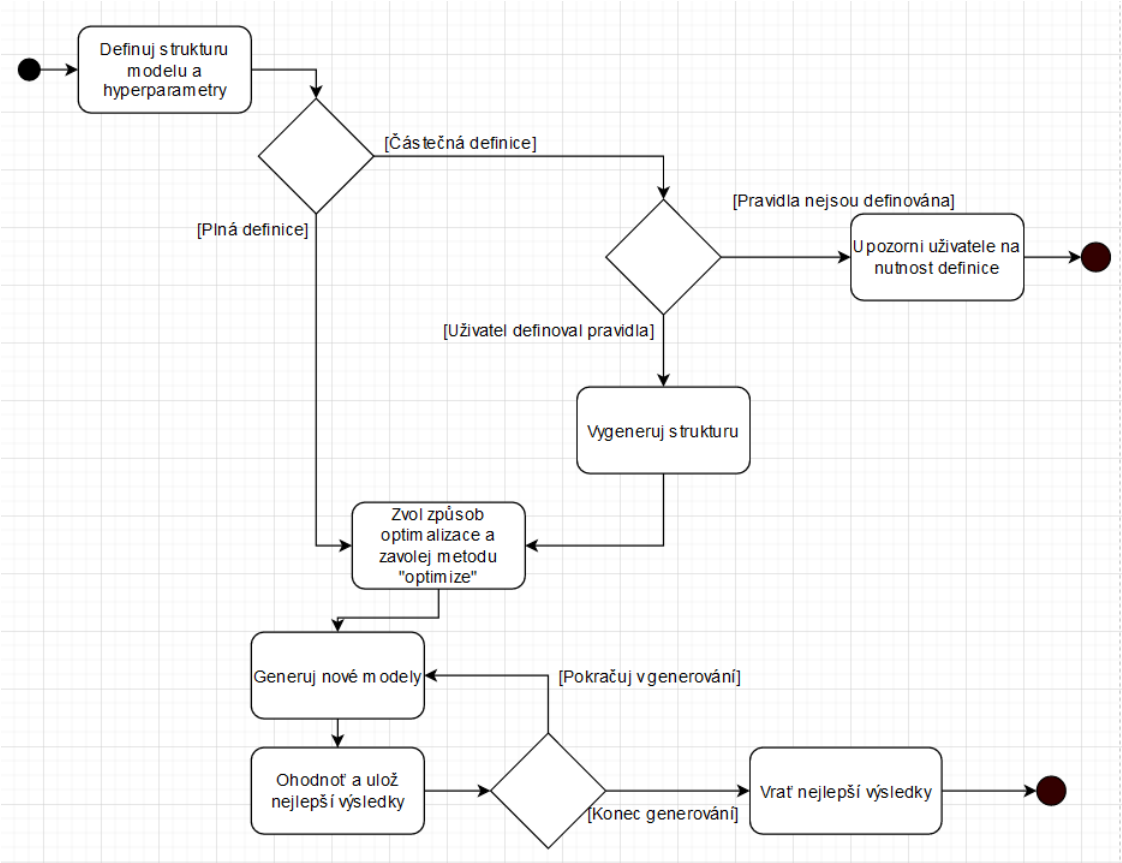
- **Float**- rozsah desetinných čísel s určitou přesností - zapisován metodou „numRange()“ s tím, že konkrétní implementaci obsahuje vlastní metoda „floatRange()“ daného pravděpodobnostního rozdělení.

- **Text** - výčet textových nastavení parametru.

3.3 Navržená funkcionalita

3.3.1 Activity diagram

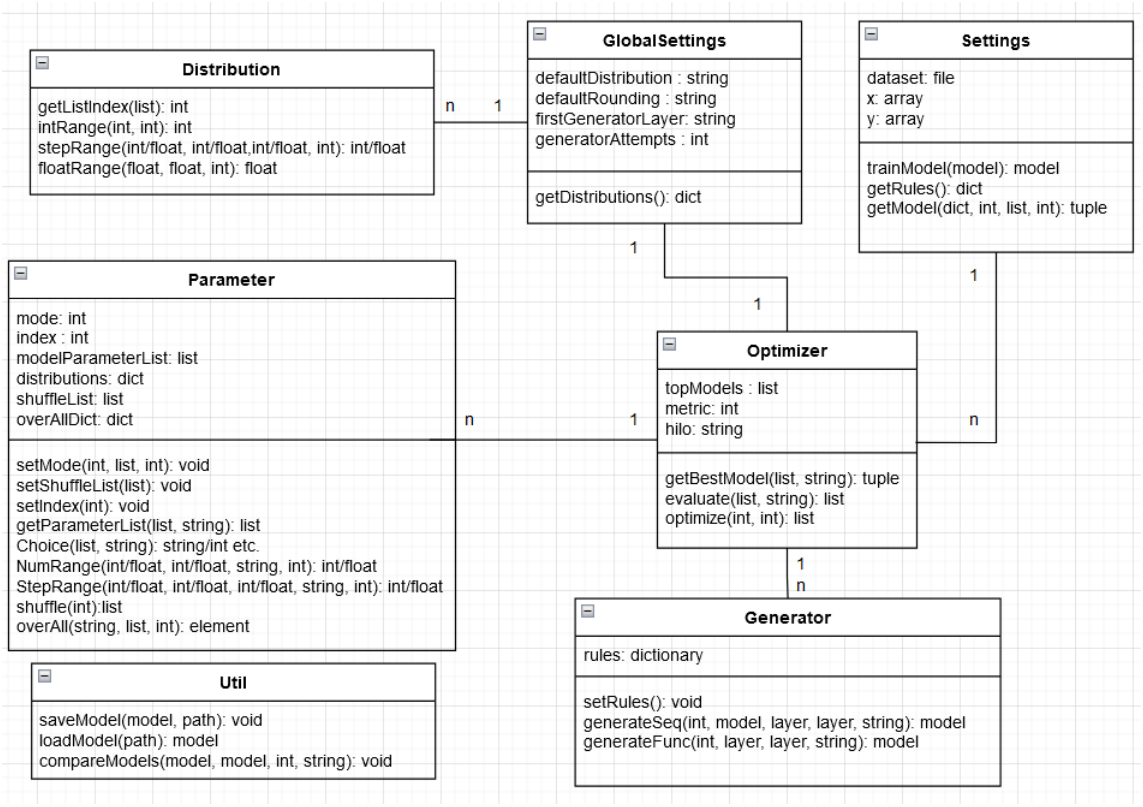
Způsob, jakým bude nové řešení fungovat je ilustrován na obrázku 3.1. Uživatel definuje nutné parametry pro spuštění a samotný model buď definuje v plném rozsahu, nebo pouze částečně - vstupní, výstupní vrstva a pravidla, pomocí kterých má být model dotvořen (například jaké vrstvy může obsahovat, rozsah modelu apod.). Poté zvolí optimalizační metodu a její konkrétní metoda „optimize“ se bude dále starat o zbytek procesu. Tato metoda bude generovat nové modely na základě konkrétní implementace a pravidel, provádět jejich evaluaci a nakonec vrátí nejlepší nalezené modely pro konkrétní problém.



Obrázek 3.1: Activity diagram

3.3.2 Class diagram

Diagram zobrazující rozložení tříd je dostupný na obrázku 3.2.



Obrázek 3.2: Class diagram

- Třída GlobalSettings: Slouží jako globální nastavení a uchování proměnných a metod potřebných pro tvorbu tříd a instancí.
  - Proměnná defaultDistribution: Definuje defaultní pravděpodobnostní rozdělení, které má být využito v případě jeho nespecifikování v konkrétních parametrech.
  - Proměnná defaultRounding: Definuje defaultní počet desetinných míst při zaokrouhlení.
  - Proměnná firstGeneratorLayer: Definuje, jaký záznam ze slovníku má být v generátoru využit jako první.
  - Proměnná generatorAttempts: Definuje maximální počet pokusů, po který se bude generátor pokoušet vytvořit funkční model na základě pravidel.
  - Metoda getDistributions: Slouží k vytvoření a vrácení slovníku obsahujícím pravděpodobnostní rozdělení, která je možná využít.
    Představíme-li si slovník s názvem distributions, poté do něj v této metodě vložíme nový záznam pomocí následujícího zápisu: „distributions[„mujNazev“] = specifickeRozdeleni()“, kdy klíč můžeme zvolit

jaký chceme (na ten se později odkazujeme, pokud volíme v nějakém parametru pravděpodobnostní rozdělení) a instanci rozdělení volíme podle toho, která rozdělení máme k dispozici (například „normalDistribution()“, implementující konkrétní metody pro normální rozdělení).

- Třída Settings: Slouží jako nastavení a uchování proměnných a metod potřebných k tvorbě a práci s modely.
  - Proměnná Dataset: Slouží k uchování datasetu, který bude využit k trénování modelu.
  - Metoda trainModel: Slouží k trénování modelu přijímaného coby parametr. V metodě následně tento natrénovaný model vracíme.
  - Metoda getRules: Slouží k vytvoření a vrácení slovníku obsahujícím pravidla pro generování vrstev v případě využití třídy „Generator“ v metodě „getModel“. Příklad využití je k dispozici v sekci Generování struktury. Tuto metodu není nutné definovat, je využita pouze v případě, že v metodě „getModel“ využíváme generování třídou „Generator“.
  - Metoda getModel: V této metodě je uživatelem definován samotný model - buďto plně, nebo částečně. Povinným parametrem metody je slovník pravděpodobnostních rozdělení a nepovinnými parametry jsou list parametrů, režim a index - tyto parametry jsou využity pouze v případě, kdy je model vytvářen z již existujího listu parametrů, které se mají dosadit, místo jejich generování. Tato metoda vrací tuple ve struktuře (model, list), obsahující samotný model a list hyperparametrů pomocí kterých byl utvořen.
- Třída Optimizer: Její zodpovědností je provedení samotné optimalizace a evaluace modelů. Tato třída slouží jako základ implementující obecné metody a parametry společné pro všechny optimizery - tyto konkrétní optimizery z ní dědí a implementují svou metodu „optimize“ takovým způsobem, který odpovídá jejich způsobu optimalizace.
  - Proměnná topModels: Slouží k uchování nejlepších modelů z cyklů optimalizace.
  - Metoda getBestModel: Slouží k získání nejlepšího modelu z listu ohodnocených modelů (z metody evaluate), který přijímá jako parametr spolu s proměnnou typu string nabývající hodnoty „high“ nebo „low“ - podle toho, zda je naším cílem danou metriku minimalizovat, či maximalizovat.

  - Metoda evaluate: Slouží k ohodnocení modelů, které přijímá jako parametr podle metriky, jejíž index je druhým parametrem (vraceným z metody „keras.evaluate“). Metoda vrací list ohodnocených modelů.
  - Metoda optimize: Provádí samotný proces optimalizace - přijímá čtyři parametry a to počet cyklů, počet generovaných modelů, index metriky a selektor. Index metriky a selektor jsou využívány k volání metod „evaluate“ a „getBestModels“. Metoda vrací model, který dosáhl v procesu nejlepšího výsledku.

- Třída Generator: Její zodpovědností je starat se o vygenerování části modelu na základě uživatelem definovaných pravidel, pokud se rozhodne ji využít (uživatel má možnost definovat celý model sám, v takovém případě je tato třída nevyužita).
  - Proměnná rules: Tato uchovává slovník pravidel, který je používám ke generování struktury. Tento slovník a příklad jsou popsány v příslušné sekci Generování struktury.
  - Metoda generateSeq: Slouží k vygenerování části sekvenčního modelu. Povinným parametrem je slovník pravidel, nepovinnými pravidly jsou model, vstup, výstup a první generovaná vrstva (záznam slovníku pravidel)- pokud jsou tyto nezadány, bude v metodě vytvořen model typu „keras.Sequential“, jejž bude metoda na konci vracet a počáteční vrstva bude použita z proměnné „globalSettings.firstGeneratorLayer“.
  - Metoda generateFunc: Slouží k vygenerování části funkcionálního modelu. Parametry jsou ”pravidla, vstupní vrstva, výstupní vrstva, první generovaná vrstva”.

- Třída Distribution: Tato třída slouží jako interface - definuje metody, které konkrétní pravděpodobnostní rozdělení implementují svým vlastním způsobem a které lze následně využít. Tyto metody momentálně jsou: „getListIndex“, „intRange“, „floatRange“, „stepRange“. Některé z metod obsahují parametr zaokrouhlení a zde je vhodné podotknout, že zaokrouhlení je nepovinným parametrem, defaultně nastavený na 2 desetinná místa. Konkrétní pravděpodobnostní rozdělení implementují tyto metody a následně je k nim přistupováno ze třídy „Parameter“.
  - Metoda getListIndex: Slouží k získání prvku z listu, který je jejím parametrem. Metoda vrací integer reprezentující daný prvek.

  - Metoda intRange: Slouží k získání celého čísla z rozsahu definovaného parametry „od, do“. Metoda vrací integer ležící v daném intervalu.
  - Metoda floatRange: Slouží k získání čísla typu float ze zadaného rozsahu definovaného parametry. Tyto parametry jsou „od, do, zaokrouhlení“. Metoda vrací číslo typu float.
  - Metoda stepRange: Slouží k získání čísla typu integer či float z rozsahu definovaného parametry. Tyto parametry jsou „od, do, krok, zaokrouhlení“. Metoda vrací číslo typu int či float.

- Třída Parameter: Tato třída slouží k samotnému označování hyperparametrů, jejich generování a ukládání pořadí parametrů při vytváření modelu. Některé z níže popsaných metod obsahují parametr zaokrouhlení a zde je vhodné podotknout, že zaokrouhlení je nepovinným parametrem, defaultně nastavený na 2 desetinná místa. Dále všechny metody (vyjma „shuffle“ a „overAll“) pracují s parametrem definovaným pravděpodobnostním rozdělením - tento parametr je rovněž nepovinný a v případě nespecifikování bude využito defaultní rozdělení.

  - Proměnná mode: Definuje režim funkcionality (0 = generování parametrů, 1 = vkládání parametrů z listu).
  - Proměnná index: Slouží jako ukazatel k pohybu ve vkládacím listu (v případě režimu generování je tato nevyužita).
  - Proměnná modelParameterList: Slouží k uchovávání pořadí parametrů právě generovaného modelu.
  - Proměnná distributions: Obsahuje dostupná pravděpodobnostní rozdělení a jejich instance, s nimiž pracuje ve svých metodách.
  - Proměnná shuffleList: Obsahuje vrstvy, jejichž pořadí není v modelu pevně dáno a může se měnit. Jedná se o list listů - můžeme takto vytvořit několik různých sad obsahujících různé vrstvy a různá nastavení, které ale budeme dosazovat pouze na daná místa.
  - Proměnná overAllDict: Obsahuje seznamy prvků, které jsou využity v metodě „overAll“.
  - Metoda setMode: Slouží k nastavení režimu pomocí parametru. V případě nastavení do režimu vkládání je nutné zadat taktéž parametry list (z nějž bude vkládání probíhat) a int (index, na kterém bude vkládání v listu začínat).

- Metoda setShuffleList: Slouží k uložení listu, který je parametrem této metody, do proměnné shuffleList.
- Metoda setIndex: Slouží k nastavení hodnoty proměnné index, který je parametrem metody.
- Metoda getParameterList: Slouží k získání obsahu proměnné „modelParameterList“ a která je defaultně volána na konci funkce „getModel“. Tato metoda vrací tuple obsahující model a list parametrů k jeho vytvoření.
- Metoda choice: Slouží k získání prvku z listu. Parametry jsou list, ze kterého vybíráme a string definující pravděpodobnostní rozdělení. Metoda následně vrací daný prvek.
- Metoda numRange: Slouží k získání čísla z rozsahu. Parametry jsou „od, do, rozdělení, zaokrouhlení“. Metoda vrací konkrétní číslo.
- Metoda stepRange: Slouží k získání čísla z rozsahu s využitím kroku. Parametry jsou „od, do, krok, rozdělení, zaokrouhlení“. Metoda vrací konkrétní číslo.
- Metoda shuffle: Tuto metodu lze využít ke vkládání prvku z listů. Například lze vkládat čísla, ale lze ji také použít například k měnění pořadí vrstev v modelu tím, že na místa svého volání vkládá vrstvy, které jsou obsaženy v listu proměnné shuffleList - v případě využití této možnosti jsou tyto vrstvy definovány uživatelem ve funkci „getModel“ a mohou obsahovat volání všech předchozích metod za účelem výběru parametrů. Obsahuje parametr definující, ze kterého listu chceme vrstvy vybírat.
- Metoda overAll: Jejími parametry jsou „identifikator (použit jako klíč ke slovníku „overAllDict“), list prvků, index (od kterého budeme začínat, nepovinný - defaultně 0)“. Metoda umožňuje průchod přes všechny prvky v listu a s každým voláním vrací prvek indexu n+1 od posledního volání.

- Třída Util: Obsahuje metody k manipulaci s modely.
  - Metoda saveModel: Slouží k uložení modelu na disk - parametry jsou uládaný model a cesta.
  - Metoda loadModel: Slouží k načtení modelu z disku - parametrem je cesta k modelu. Tento model metoda následně vrací.

– Metoda compareModels: Slouží k porovnání dvou modelů (například jeden převzatý, druhý vytvořený pomocí optimalizace). Parametry jsou „první model, druhý model, index metriky, hodnota (high/low)“.

Třídy jsou v novém řešení navrženy takovým způsobem, aby bylo možné je do budoucna snadno rozšířit i o nová pravděpodobnostní rozdělení a způsoby optimalizace, jaké může daný uživatel požadovat. Pro zavedení nového pravděpodobnostního rozdělení stačí definovat příslušné metody podle třídy „Distribution“, obdobně pro způsob optimalizace stačí v novém definovat metodu „optimize“.

3.4 Generování struktury

3.4.1 Třída Generator

Nové řešení podporuje možnost nechat třídu „Generator“ vytvořit část modelu. Jak je zmíněno výše, třída „Generator“ obsahuje vlastní metodu pro sekvenční i funkcionální Keras model. Tyto metody dokáží přidat zadané množství vrstev za sebe, přičemž tyto vrstvy jsou vybírány na základě pravidel. Vhodná metoda je volána s příslušnými vstupy ve třídě „getModel“ na příslušném místě, kde chceme generátor použít (můžeme jej využít na několika místech zároveň).

3.4.2 Pravidla

K definici pravidel je využita struktura dictionary, kterou jazyk Python obsahuje. Představme si, že máme vytvořen prázdný slovník označený pro přehlednost „rules“. Do tohoto slovníku budeme vkládat ve funkci „getRules“ pravidla pro generování. Jako příklad definování pravidla můžeme uvést následující zápis:
„rules[„dense“]=[tf.keras.Dense(32, activation=„relu“), [„dense“, „vrstva1“]]“.
Tímto zápisem uložíme vrstvu pod klíčem „dense“, kdy prvním prvkem je zápis dané vrstvy způsobem, jakým má být vytvořena a druhým prvkem je list obsahující záznamy slovníku (tedy námi definovaná pravidla), které mohou následovat v modelu po této konkrétní vrstvě. V tomto zápisu je rovněž možno vytvořit a následně využít instanci třídy „Parameter“(pro následující část je její instance nazvána p) a některé části parametrizovat - příkladem může být zápis:
„rules[vrstva] = [tf.keras.Dense(p.Choice([2,4,8])), [„vrstva1“]]“.

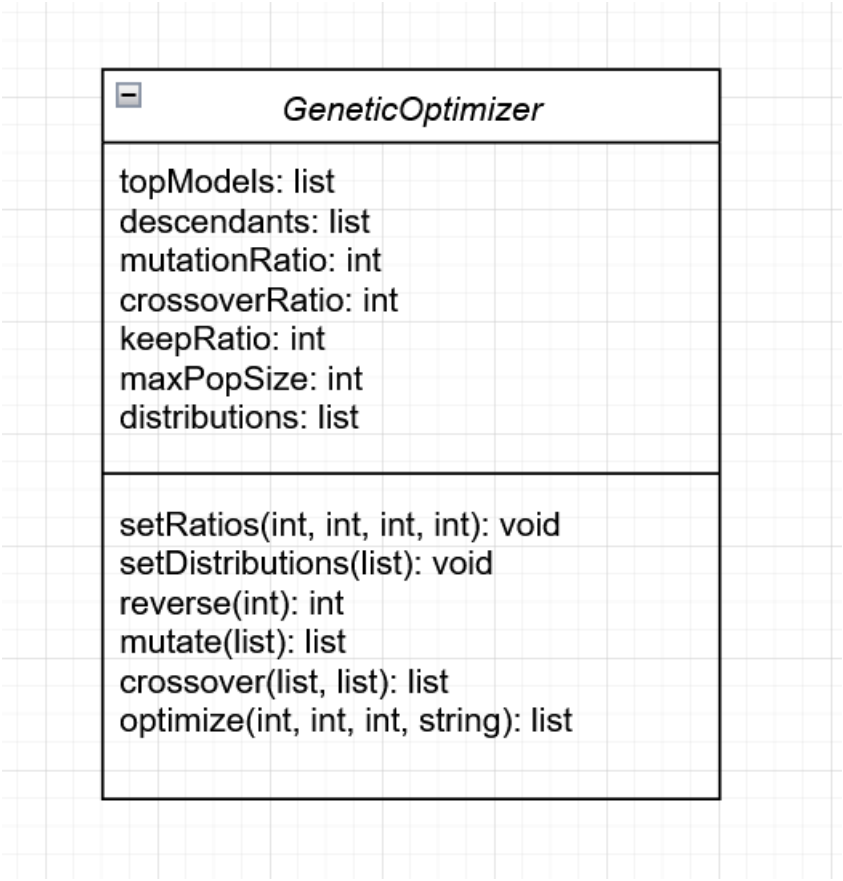
3.5 Optimalizační metody

3.5.1 Náhodný výběr

První podporovanou metodou nového řešení je náhodná volba. Všechny parametry jsou v každém vytvořeném modelu voleny náhodně a tento proces probíhá po parametricky zadaný počet cyklů v nichž jsou generovány modely do zadaného počtu - tyto modely jsou s koncem každého cyklu evaluovány a nejlepší z nich je uložen do struktury list. S tímto je po konci procesu možno pracovat - vzít libovolný model z procesu, například ten zcela nejlepší a následně jej používat dále, případně jej uchovat na disku pro pozdější použití.

3.5.2 Genetická optimalizace

Zajímavější metodou je druhá, jejíž třída je zobrazena na obrázku 3.3 a která je založena na principu fungování genetických algoritmů - to jsou algoritmy inspirované biologií a jejím poznáním genetiky využívající se k řešení obtížných problémů, a která k nalezení nejlepšího řešení využívají několik metod pomocí kterých dochází k šlechtění populace tvořenou množstvím jedinců (v našem případě modely). Tato populace je zprvu vytvořena náhodně a následně jsou na ni aplikovány zmíněné metody, čímž dochází k jejímu šlechtění za účelem „vypěstovat“ nejlepší výsledek. Tyto metody jsou „mutace“, kdy dojde ke změně kódu, kterým je jedinec tvořen (v našem případě jde o list parametrů, které byly použity při jeho vytvoření v metodě „getModel“) v některém místě kódu, dále „křížení“ - například vezmeme dva jedince a vytvoříme z nich nového způsobem, že tento potomek je složen ze dvou částí kódu získaných od jeho rodičů, tedy původních modelů. Tyto metody jsou v určité míře aplikovány na populaci v každém cyklu a následně jsou zvoleni jedinci, kteří budou nadále její součástí, můžeme říci, že tito přežijí a ostatní pozice v populaci budou doplněny nově vytvořenými jedinci. Na konci každého cyklu ukládáme nejlepšího jedince do specifického listu. Po skončení procesu bude list obsahovat nejlepší jedince ze všech cyklů a s nimi bude možné dále pracovat - například nejlepšího z nich uchovat na disku pro pozdější použití.



Obrázek 3.3: Třída genetického optimizeru

- Proměnná topModels: Účelem této proměnné typu list je udržovat nejlepší modely ze všech cyklů.
- Proměnná descendants: Účelem této proměnné je dočasně udržet modely, které mají být zachovány do následující generace po skončení jednoho cyklu.
- Proměnná mutationRatio: Účelem této proměnné je definovat množství mutací, ke kterým dojde v populaci na konci jednoho cyklu.
- Proměnná crossoverRation: Účelem této proměnné je definovat množství křížení, ke kterým dojde v populaci na konci jednoho cyklu.
- Proměnná keepRatio: Účelem této proměnné je definovat množství modelů, které má být zachováno do následující generace po konci jednoho cyklu.
- Proměnná maxPopSize: Účelem této proměnné je definovat maximální velikost populace, kterou můžeme v procesu dosáhnout.
- Proměnná distributions: Účelem je uchování slovníku s rozděleními, ze kterého můžeme čerpat.

- Metoda setRatios: Nastavuje hodnoty proměnných v pořadí: „mutationRatio“, „crossoverRatio“, „keepRatio“ a „maxPopSize“.
- Metoda setDistributions: Umožňuje nastavit hodnotu proměnné distributions na slovník předávaný jako parametr.
- Metoda reverse: Vstupem je číslo 0 či 1 a metoda vrací jeho opak - slouží k nastavení příznaku, zda má být daný parametr změněn, či nikoliv.
- Metoda mutate: Její zodpovědností je pozměnit kód tvořící daného jedince, který je metodě předán coby parametr. Metoda následně vrací pozměněný kód v podobě listu.
- Metoda crossover: Její zodpovědností je náhodné zkřížení kódu dvou jedinců, které jsou předány coby parametry. Metoda následně vrací kód tvořící nového jedince.
- Metoda optimize: Samotná řídící metoda, jejímiž parametry jsou dvě čísla, z nichž první určuje počet cyklů a druhé velikost populace. Její zodpovědností je vytvoření a udržování populace za aplikace výše zmíněných metod a generování nových jedinců. Metoda následně vrací list nejlepších modelů, které se za daný proces podařilo najít v daných cyklech.

3.6 Využití

3.6.1 Způsob užití

Toto řešení je zamýšleno pro uživatele, kteří hledají nástroj umožňující testování různých nastavení parametrů modelů. Oproti již existujícím řešením odstraňuje několik nedostatků. Taktéž umožňuje pro přehlednost označovat parametry přímo na místech, kam mají být dosazovány, umožňuje práci s pravděpodobnostními rozděleními ve všech metodách, kde toto využití dává smysl a následně hledat optimální nastavení a získat tak nejlepší nalezený model pomocí zvolené optimalizační metody - optimalizaci založenou na genetických algoritmech jiný nástroj nenabízí. Řešení je možné do budoucna snadno rozšiřovat o nová rozdělení a optimalizační metody implementací vyžadovaných metod podle tříd „Distribution“ a „Optimizer“.

Parametry

Tato část je již popsána v dokumentaci třídy „Parameter“, rád bych ale v této konkrétní sekci uvedl několik příkladů užití a volání příslušných metod. Pro jednoduchost předpokládejme instanci třídy „Parameter“ pojmenovanou p.

- Zápis „p.Choice([1,2,3,4,5], „normal“)“ - vrátí prvek listu (nemusí být pouze číselná hodnota) s tím, že výběr bude proveden v normálním rozdělení („normal“ je zde i dále klíč slovníku rozdělení definovaný v metodě „getDistributions“ třídy „Settings“- názvy i nastavení rozdělení lze měnit v této metodě).

- Zápis „p.numRange(0, 10)“ - vrátí číslo z daného intervalu.

- Zápis „p.shuffle(1)“ - na místo volání této metody dosadí prvek z n-tého listu (parametr metody, v tomto případě tedy ten druhý - počítáno od nuly) v proměnné ”shuffleList”(struktura list obsahující listy, ze kterých vkládáme).

- Zápis „p.overAll(„Oznaceni1“, [1,2,3,4,5], 4)“ - prochází přes všechny prvky a s každým voláním této metody s daným označením (v tomto případě „Oznaceni1“) vrací prvek indexu n+1 od minulého volání. Třetím prvkem je index, od kterého chceme průchod začít, pokud není uveden, použijeme 0, tedy od prvního prvku.

Generování

Stejně jako výše parametry, rád bych zde uvedl způsob zápisu a volání metod třídy „Generator“. Pro jednoduchost předpokládejme instanci třídy „Generator“ pojmenovanou g.

- Zápis „g.generateSeq(5, model, defLayer=„dense“)“ je využívám v sekvenčních modelech. Vstupy jsou „rozsah, pravidla, model“ s tím, že model je obohacován o nové vrstvy dle definovaných pravidel. Tento model je následně metodou vracen. Využití v kontextu je dostupné níže v sekci Praktická ukázka.

- Využití funkcionální metody, kdy je vygenerován zadaný počet vrstev lze předvést na následujícím zápisu: „x = g.generateFunc(1, model)“ a následně můžeme tuto strukturu posílat dále dle principů funkcionálního API pro ukázku pomocí zápisu „output = layers.Dense(10)(x)“.

3.6.2 Praktická ukázka

Praktické využití aplikace je demonstrováno na jednoduchém datasetu a dl modelu inspirovaným článkem dostupným z [7]. Jedná se o dataset „Pima Indians Diabetes“, jehož obsahem jsou data o lidech v devíti sloupcích, přičemž prvních osm jsou lékařské parametry osoby (například krevní tlak, věk ...) a poslední definuje, zda byl daný člověk nakažen, či nikoliv. Definice modelu spolu se spuštěním procesu za využití genetického optimizeru je demonstrována na obrázku 3.4, přičemž proces je popsán níže. Instance genetického optimizeru je zde vytvářena s několika parametry, které ovlivňují jeho samotný proces. Těmito parametry jsou v pořadí „mR“ - počet mutací prováděných na populaci, „cR“ - počet křížení mezi členy populace, „kR“ - počet nejlepších modelů zachovávaných do následující generace, „pS“ - maximální možná velikost populace. Instance této třídy může být vytvořena i bez těchto parametrů, přičemž v takovém případě se využije defaultní nastavení. Nastavení s daty je dostupné v příloze 2 kapitoly Přílohy.

```
    def getModel(d, mode = 0, lst = [], n=0):
        t = Parameter(mode,lst, d)
        g = Generator()

        model = keras.Sequential()
        model.add(layers.Dense(12, input_dim=8, activation='relu'))

        model = g.generateSeq(t.NumRange(0,6), model)

        model.add(layers.Dense(1, activation='sigmoid'))

        return (model, t.getParameterList())
```

```
g = GeneticOptimizer(mR = 5, cR = 2, kR = 4, pS = 20)
m = g.optimize(20,30,1,"high")
```

Obrázek 3.4: Příklad použití - zápis

V ukázce je v metodě „getModel“ definován jednoduchý sekvenční model. O správu parametrů se zde stará instance třídy „Parameter“, zde označena jako „t“. Samotné označení parametru je prováděno přímo na místech, kam má být daný parametr vložen - zde ukázáno při použití generátoru, kdy počet generovaných vrstev je parametrizován zápisem „t.NumRange(0,6)“ - značícím, že množství vygenerovaných vrstev bude v tomto rozsahu. Metodu generátoru pro generování struktury sekvenčních modelů zde voláme zápisem:
„g.generateSeq(t.NumRange(0, 6), model)“, kdy parametry jsou zde rozsah generování a model (náš sekvenční model, do kterého budou vrstvy přidány). Zde

je vhodné zmínit pravidla generátoru, která v této ukázce obsahují dvě možné vrstvy, ze kterých se může vytvořená struktura skládat. Tato pravidla jsou definována v metodě „getRules“ našeho nastavení. Ukázka nastavení využitého v tomto příkladu je dostupná na obrázku 3.5.

```
def getRules(): #definice slovníku s pravidly
    p = Parameter()
    rules = {}

    rules["dense"]=(tf.keras.layers.Dense(p.Choice([4,8]), activation="relu"),["dense", "drop"])
    rules["drop"]=(tf.keras.layers.Dropout(0.2, noise_shape = None, seed = None),["dense"])

    return rules
```

Obrázek 3.5: Příklad použití - definice pravidel

Metoda „getModel“ je pouze součástí nastavení, samotné spuštění procesu je vykonáváno o buňku níže, kde vytváříme instanci třídy „geneticOptimizer“, zde označenou jako „g“ a o řádek níže (tedy „g.optimize(20, 30, 1, „high“)“) voláme metodu optimize s parametry 20 - počet cyklů (vytvářených generací), 30 - velikost naší populace, 1 - index metriky z metody „model.evaluate“ (v našem případě se jedná o „accuracy“) a „high“ - značí, že metriku chceme maximalizovat. Samotný proces po spuštění vytváří několik výpisů sloužících k informování uživatele - který cyklus právě probíhá a hodnotu metriky nejefektivnějšího modelu v daném cyklu. Po dokončení všech cyklů proces vypíše hodnotu metriky nejlepšího nalezeného modelu a tento vrátí - zde ukládán do proměnné „m“, protože s ním chceme dále pracovat. Na obrázku 3.6 je k vidění struktura nejlepšího modelu, který proces vybral - tato struktura je dostupná pod metodou keras modelů „summary“. Pokud bychom chtěli pracovat i s jiným z modelů, nikoliv pouze nejlepším, je možné tento získat pomocí proměnné „g.bestModels“.Struktura této proměnné obsahuje jak modely, tak hodnoty jejich parametrů i metriku, je proto snadné se v ní orientovat. S danými modely můžeme dále pracovat například pomocí metody „saveModel“ třídy „Util“, která by tento model uložila na disk.

```
In [5]: m.summary()
```

```
Model: "sequential_185"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1036 (Dense)           (None, 12)                108
_________________________________________________________________
dense_1037 (Dense)           (None, 4)                 52
_________________________________________________________________
dense_1038 (Dense)           (None, 8)                 40
_________________________________________________________________
dense_1039 (Dense)           (None, 4)                 36
_________________________________________________________________
dense_1041 (Dense)           (None, 1)                 5
=================================================================
```

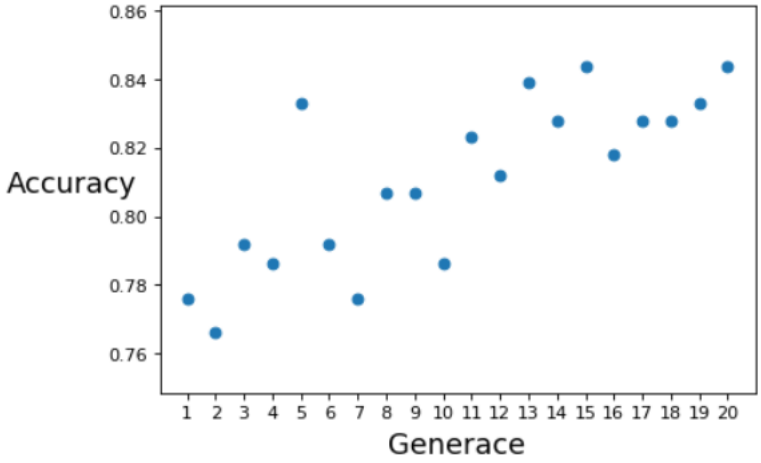
Obrázek 3.6: Příklad použití - struktura nejefektivnějšího modelu

Mé řešení podporuje mimo sekvenční modely taktéž možnost využít funkcionální API. Pro účel demonstrace je na obrázku 3.7 uvedena ukázka tvorby funkcionálního modelu s využitím příslušné metody generátoru, která vytváří strukturu pro funkcionální modely - tedy metoda „generateFunc“.

```
def getModel(d, mode = 0, lst = [], n=0): #definice modelu
    t = Parameter(mode,lst, d) #instance pro tvorbu a označování parametrů
    g = Generator() #pro případ využití generování části struktury

    inputs = keras.Input(shape=(8,))
    x = layers.Dense(12, input_dim=8, activation='relu')(inputs)

    x = g.generateFunc(t.NumRange(0,6), x)
    outputs = layers.Dense(1, activation='sigmoid')(x)

    model = keras.Model(inputs=inputs, outputs=outputs, name="test_model")


    return (model, t.getParameterList()) #vracíme tuple - model a list parametrů
```

Obrázek 3.7: Příklad použití - definice pomocí funkcionálního API

Výsledky experimentu zobrazující hodnotu metriky „accuracy“ nejlepších nalezených modelů v závislosti na generaci je dostupný na obrázku 3.8.



Obrázek 3.8: Graf přesnosti řešení v závisosti na generaci.

Z grafu je patrná rostoucí tendence naší metriky se zvyšujím se počtem generací. Cílem tohoto experimentu bylo nalezení modelu a jeho struktury, který bude na testovacích datech dosahovat co nejvyšší hodnoty metriky „acccuracy“. V zde uvedeném případě byl zcela nejefektivnější model nalezen v patnácté generaci s hodnotou metriky 84.4%.

3.7 Srovnání s existujícími řešeními

Úvodem ke srovnání bych zde chtěl zmínit, že následující srovnání je provedeno vůči řešením, která podporují sekvenční i funkcionální Keras model, tedy řešení Talos a Keras-Tuner. K otestování funkcionality řešení jsem využil tři datasety, z nichž prvním je „Iris“ - tento dataset obsahuje data o kosatcích v pěti sloupcích, přičemž první čtyři obsahují údaje o konkrétních rostlinách (například délka okvětního lístku) a poslední definuje její přesný druh ze třech kategorií. Druhým datasetem je „Pima Indians Diabetes“ využitý taktéž v praktické ukázce výše. Tento dataset obsahuje data o lidech v devíti sloupcích, přičemž prvních osm jsou lékařské parametry osoby (například krevní tlak, věk ...) a poslední definuje, zda byl daný člověk nakažen, či nikoliv. Třetím datasetem je dataset „Mnist“, který obsahuje velké množství ručně psaných číslic rozdělených do trénovací a testovací množiny.

Problém jsem v nabízených řešeních implementoval s co nejvyšší úrovní podobnosti a následně stejný problém otestoval taktéž s využitím generování vrstev, které mé řešení nabízí. Nastavení na úrovni možností existujících řešení pro dataset „Iris“ je dostupné ve formě pseudokódu na obrázku 3.9 s tím, že parametry

jsou v obrázku označeny na místech, kam budou dosazovány.

```
Sequential()
Dense(32, input_dim=4, activation = Choice(["relu", "elu"]))
Dense(3, activation = "softmax")

compile(optimizer = Choice(["Nadam", "Adam"]), loss = "categorical_crossentropy",
        metric = ["accuracy"])

fit(x_train, y_train,
   batch_size = Choice([20,30,40]),
   epochs = Choice([10,20]))
```

Obrázek 3.9: Dataset Iris - nastavení

Druhý pseudokód s nastavením obsahujícím generování vrstev a pravidla generování je dostupný na obrázku 3.10, přičemž zde pravidla definují možnost generování struktury obsahující jeden druh vrstvy, ve které bude množství neuronů náhodně vybíráno z rozsahu 8 - 20 s krokem čtyři (tedy může zde být dosazováno 8, 12 atd.). Pravidla rovněž definují, že vrstvu „dense“ může následovat pouze další vrstva „dense“.

```
rules: ["dense"]=(Dense(p.StepRange(8,20,4), activation="relu"),["dense"])

Sequential()
Dense(32, input_dim=4, activation = Choice(["relu", "elu"]))
generateSeq(NumRange(0,5))
Dense(3, activation = "softmax")

compile(optimizer = Choice(["Nadam", "Adam"]), loss = "categorical_crossentropy",
        metric = ["accuracy"])

fit(x_train, y_train,
   batch_size = Choice([20,30,40]),
   epochs = Choice([10,20]))
```

Obrázek 3.10: Dataset Iris - nastavení s přidáním generátoru

K dosažení největší podobnosti nastavení bylo u řešení Keras-Tuner ještě nutné specifikovat hodnoty „batch size“ a „epochs“ definováním vlastního tuneru dle návodu v dokumentaci. Po spuštění všech řešení jsem mohl získat konkrétní hodnoty metriky „accuracy“, přičemž cílem bylo získat její co nejvyšší hodnotu.

Pro dataset „Iris“ dokázalo mé řešení za využití generátoru najít model dosahující 100% „accuracy“ na testovacím datasetu již ve druhé generaci. Spuštění, výstup a struktura nejlepšího nalezeného modelu je dostupná na obrázku 3.11. Na základě výstupu lze konstatovat, že vytvořením dodatečné struktury bylo možné značně zvýšit přesnost modelu.

```
g = GeneticOptimizer()
m = g.optimize(5,10,1,"high")
```

```
Cyklus:  1  /  5
Nejlepsi model cyklu dosahl hodnoty metriky:  0.947
Cyklus:  2  /  5
Nejlepsi model cyklu dosahl hodnoty metriky:  1.0
Cyklus:  3  /  5
Nejlepsi model cyklu dosahl hodnoty metriky:  1.0
Cyklus:  4  /  5
Nejlepsi model cyklu dosahl hodnoty metriky:  1.0
Cyklus:  5  /  5
Nejlepsi model cyklu dosahl hodnoty metriky:  1.0
Nejlepsi model v celkovem procesu dosahl hodnotu metriky: 1.0
```

```
m.summary()
```

```
Model: "sequential_169"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1130 (Dense)           (None, 32)                160
_________________________________________________________________
dense_1131 (Dense)           (None, 16)                528
_________________________________________________________________
dense_1133 (Dense)           (None, 3)                 51
```

Obrázek 3.11: Dataset Iris - spuštění, výstup a struktura

Prakticky stejným způsobem jsem postupoval také s datasetem „Pima Indians Diabetes“, jehož nastavení je ve formě pseudokódu dostupné na obrázku 3.12. Nejefektivnějším se zde ukázal model, jehož struktura je dostupná na obrázku 3.6. Vrstvy 2-4 byly v tomto modelu vytvořeny pomocí generátoru.

```
Sequential()
Dense(12, input_dim=8, activation = "relu")
Dense(Choice([4, 8]), activation = 'relu')
Dense(1, activation = "sigmoid")

compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']

fit(x_train, y_train,
   batch_size = 10,
   epochs = 25)
```

Obrázek 3.12: Dataset Indians diabetes - nastavení

Druhý pseudokód s nastavením obsahujícím generování vrstev a pravidla generování je dostupný na obrázku 3.13, přičemž zde pravidla definují možnost generování struktury obsahující dvě vrstvy, přičemž bude množství neuronů vrstvy „Dense“ voleno z hodnot 4 a 8. V pravidlech je rovněž definováno, že vrstvu „dense“ může následovat vrstva „dense“ a „drop“, přičemž vrstvu „drop“ může následovat pouze vrstva „dense“.

```
rules: ["dense"]=(Dense(Choice([4,8]), activation="relu"),["dense", "drop"]),
       ["drop"]=(Dropout(0.2, noise_shape = None, seed = None),["dense"])

Sequential()
Dense(12, input_dim=8, activation = "relu")
generateSeq(NumRange(0,6))
Dense(1, activation = "sigmoid")

compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']

fit(x_train, y_train,
   batch_size = 10,
   epochs = 25)
```

Obrázek 3.13: Dataset Indians diabetes - nastavení s přidáním generátoru

Obdobným způsobem jsem postupoval také s datasetem „Mnist“, jehož nastavení je ve formě pseudokódu dostupné na obrázku 3.14.

```
Sequential()
Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
       input_shape=(28, 28, 1))
MaxPooling2D((2, 2))
Flatten()
Dense(100, activation='relu', kernel_initializer='he_uniform')
Dense(10, activation='softmax')

compile(optimizer=SGD(learning_rate=StepRange(0.001, 0.01, 0.003), momentum=0.9),
              loss='categorical_crossentropy', metrics=['accuracy'])

fit(x_train, y_train, batch_size=128, epochs=5)
```

Obrázek 3.14: Dataset Mnist - nastavení

Druhý pseudokód s nastavením obsahujícím generování vrstev a pravidla generování je dostupný na obrázku 3.15, přičemž zde pravidla definují možnost generování struktury obsahující jeden druh vrstvy, ve které bude množství neuronů náhodně vybíráno z rozsahu 32- 64 s krokem 8. Pravidla rovněž definují, že vrstvu „conv“ může následovat pouze další vrstva „conv“.

```
rules: ["conv"]=(Conv2D(StepRange(32, 64, 8), (3, 3), activation='relu',
                        kernel_initializer='he_uniform'),["conv"])

Sequential()
Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform',
       input_shape=(28, 28, 1))
MaxPooling2D((2, 2))
generateSeq(NumRange(0,4))
Flatten()
Dense(100, activation='relu', kernel_initializer='he_uniform')
Dense(10, activation='softmax')

compile(optimizer=SGD(learning_rate=StepRange(0.001, 0.01, 0.003), momentum=0.9),
              loss='categorical_crossentropy', metrics=['accuracy'])

fit(x_train, y_train, batch_size=128, epochs=5)
```

Obrázek 3.15: Dataset Mnist - nastavení s přidáním generátoru

Po spuštění se jako nejefektivnější ukázal model o struktuře dostupné na obrázku 3.16, kdy vrstvy 3 a 4 byly vytvořeny generátorem.

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_14 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 11, 11, 32)        9248
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 9, 9, 64)          18496
_________________________________________________________________
flatten_5 (Flatten)          (None, 5184)              0
_________________________________________________________________
dense_10 (Dense)             (None, 100)               518500
_________________________________________________________________
dense_11 (Dense)             (None, 10)                1010
_________________________________________________________________
```

Obrázek 3.16: Dataset Mnists - nalezená struktura

Konkrétní číselné výsledky experimentů a hodnoty metriky „accuracy“, kterých daná řešení dosáhla u svých nejlepších nalezených modelů na testovacích datech, jsou uvedeny v tabulce na obrázku 3.17. Sloupce zde udávají použitá řešení a řádky využitý dataset. Průnik obsahuje konkrétní dosažené hodnoty nejlepšího získaného modelu v procentech.

Accuracy:	Talos	Keras-Tuner	Mé řešení	Generator
Iris	98%	95.3%	95.8%	100%
Pima indians	70%	79%	79.5%	84.4%
Mnist	98.2%	98.5%	98.3%	99%

Obrázek 3.17: Tabulka s výsledky

Na základě provedených experimentů a konkrétních výstupů mohu říci, že novinka představená v mém řešení ve formě možnosti generování struktury umožňuje najít model se strukturou dosahující efektivnějších výsledků, než modely nalezené ostatními řešeními.

Závěr

Účelem této práce bylo vytvořit nástroj schopný optimalizovat parametry a strukturu deep learning modelů. Nástroj je navržen takovým způsobem, aby jej bylo do budoucna možné rozšířit o další pravděpodobnostní rozdělení a metody optimalizace.

Na základě rešerše existujících řešení byla navržena funkcionalita schopna překonat možnosti existující řešení. Samotné parametry jsou označovány přímo v definici modelu a vytvořený nástroj rovněž nabízí oproti existujícím řešením několik nových možností parametrizace. Zde představené řešení umožňuje práci s pravděpodobnostními rozděleními a jejich volby přímo při definování parametru bez nutnosti volání různě pojmenovaných funkcí dělajících totéž, jen s využitím odlišných rozdělení.

Z hlediska způsobu optimalizace tento nástroj nabízí možnost optimalizace parametrů pomocí náhodné volby a taktéž možnost využití nové optimalizační metody založené na genetických algoritmech.

Hlavní novinkou na poli práce s modely je zde představená možnost využití generování struktury jak pro sekvenční, tak pro funkcionální API za využití příslušných funkcí generátoru, přičemž samotná struktura modelu je tímto způsobem měněna na základě definovaných pravidel.

Možnost generování struktury se při testování ukázala jako způsob umožňující automatizované získání modelů dosahujících lepších výsledků, než ostatní řešení, která toto neumožňují. Na základě testování a představené funkcionality mé řešení tedy překonává ta již existující. Výstupem práce je funkční a použitelný nástroj splňující cíle zadání.

Seznam použité literatury

[1] CHOLLET, François. Deep learning v jazyku Python: knihovny Keras, Tensorflow. Praha: Grada Publishing, 2019. Knihovna programátora (Grada). ISBN 978-80-247-3100-1.

[2] Hyperparameter tuning with Keras and Ray Tune [online]. [cit. 2021-03-18]. Dostupné z: https://towardsdatascience.com/hyperparameter-tuning-with-keras-and-ray-tune-1353e6586fda

[3] Keras dokumentace [online]. [cit. 2020-12-23]. Dostupné z: https://keras.io/

[4] Talos dokumentace [online]. [cit. 2021-03-18]. Dostupné z: https://github.com/autonomio/talos/

[5] Keras-Tuner dokumentace [online]. [cit. 2021-03-18]. Dostupné z: https://keras-team.github.io/keras-tuner/

[6] Hyperas [online]. [cit. 2021-10-14]. Dostupné z: https://maxpumperla.com/hyperas/

[7] Your First Deep Learning Project in Python with Keras Step-By-Step [online]. [cit. 2021-11-20]. Dostupné z: https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/

[8] Minimal Talos example [online]. [cit. 2021-11-27]. Dostupné z: https://gist.github.com/mikkokotila/4c0d6298ff0a22dc561fb387a1b4b0bb

Přílohy

Přílohy jsou dostupné v přiloženém komprimovaném souboru.

1. Přehled nejdůležitějších metod a stručný návod k použití.
2. Spustitelný kód řešení v jupyter notebooku spolu s datasetem využitým k demonstraci funkcionality.