Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Fizyki i Informatyki Stosowanej

nformatyki Stosowanej i Fizyki Komputerowej

Praca dyplomowa
inżynierska

Interfejs wykorzystania modeli sieci neuronowych w aplikacji Unity3d

Interface for neural network models in Unity3d application

Imię i nazwisko:	Michał Pabjan
Kierunek studiów:	Informatyka Stosowana
Opiekun pracy:	dr inż. Bartłomiej Rachwał

Kraków, styczeń 2021

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

.....
(czytelny podpis)

Na kolejnych dwóch stronach proszę dołączyć kolejno recenzje pracy popełnione przez Opiekuna oraz Recenzenta (wydrukowane z systemu MISIO i podpisane przez odpowiednio Opiekuna i Recenzenta pracy). Papierową wersję pracy (zawierającą podpisane recenzje) proszę złożyć w dziekanacie celem rejestracji.

Streszczenie

Głównym celem projektu inżynierskiego opisanego w niniejszej pracy dyplomowej było stworzenie generycznego interfejsu dla modeli sieci neuronowych pod aplikacje rozwijane w Unity 3D - silniku do tworzenia trójwymiarowych oraz dwuwymiarowych gier komputerowych lub innych materiałów interaktywnych, takich jak wizualizacje czy animacje. Praca ma charakter przeglądowy, a jego finalny produkt to aplikacja rozszerzonej rzeczywistości do detekcji oraz rozpoznawania obiektów w czasie rzeczywistym, opracowana przy współpracy z Panem D. Kobyra. Aplikacja ta, oraz jej pierwowzory, posłużyły jako środowisko eksperymentalne, na którego podstawie testowano dostępne metodyki wdrażania i konfigurowania modeli sztucznej inteligencji w Unity.

Spis treści

1 Wprowadzenie **6**
1.1 Problematyka pracy 6
1.2 Zawartość pracy 7

2 Zewnętrzne modele ML w Unity **9**
2.1 TensorFlow 9
2.2 TensorFlowSharp 13
2.3 Open Neural Network Exchange 17
2.4 Biblioteka inferencyjna - Unity Barracuda 18
2.5 Barracuda a TensorFlowSharp 24

3 Wewnętrzne mechanizmy ML w Unity **25**
3.1 Unity ML Agents 25
3.2 Unity ML Agents a Barracuda 28

4 Silniki Inferencyjne a problemy inżynierskie **30**
4.1 Przykładowe problemy i rozwiązania 30
4.1.1 Klasyfikacja obrazu 31
4.1.2 Przechwytywanie ruchu 34
4.2 Trendy ML w Unity 35
4.2.1 Virtual Youtubers (vTubers) 35

5 Aplikacja AR oparta o silnik inferencyjny Barracuda **38**
5.1 Interfejs dla modeli YOLOv2-tiny przy uzyciu Barracuda 38

6 Podsumowanie **43**
6.1 Wnioski 43
6.2 Perspektywy rozwoju 44

Rozdział 1

Wprowadzenie

1.1 Problematyka pracy

Zapotrzebowanie na systemy sztucznej inteligencji wydaje się rosnąć, a słowo uniwersalność pada coraz to częściej. Modele zachowań inteligentnych są uważane za zdolne do realizacji wybranych funkcji umysłu i ludzkich zmysłów. Można więc uznać że dowolny model ML - (ang. machine learning'owym), to niejako umiejętność którą w większości każdy z nas potrafi wykonać, a ludzki mózg to poniekąd system, który te umiejętności przechowuje. Na gruncie tej analogii, pojawia się pytanie, jak ukształtować system, jak gdyby sztuczny mózg, który posługiwałby się konkretną umiejętnością, modelem ML , zależnie od potrzeby. Dla przykładu, podczas gry w szachy, wykorzystujemy umiejętność gry w szachy, a podczas robienia herbaty, umiejętności robienia herbaty, ów dobór umiejętności do otoczenia wydaje się bardzo naturalny mimo że owe czynności nie mają wiele ze sobą wspólnego, natomiast hipotetyczne modele funkcjonujące w sposób pozwalający realizować te akcje, posiadały by znacznie odmienne wejścia i wyjścia. Zatem, cel stworzenia mechanizmu, który byłby swego rodzaju interfejsem dla większej ilości modeli o odrębnych funkcjonalnościach, obecnie wydaje się być abstrakcyjny. Niemniej jednak ukształtowanie interfejsu dla modeli ML, ale o zbliżonych stosowalnościach, zdaję się być bliższe naszemu wyobrażeniu. Przykładowo, na obrazie pochodzącym z dowolnie przyjętych receptorów rejestrujących obszar znajdujący się poza pojazdem podczas jego przemieszczania się, dostosowany model wykrywałby by otoczenie związane z ruchem drogowym, z kolei w trakcie przebywania w budynku, obraz z tego samego źródła zostałby analizowany przez model dostosowany do otoczenia wnętrz budynków, w obu scenariuszach wykorzystując ten sam system, który w takim przypadku stałby się łącznikiem dla dwóch

modeli ML służących do detekcji obiektów. Taki system posiadałby możliwość dostosowywania się do otoczenia. Przykładowo model wykrywający przeszkody na drodze powinien niezwłocznie reagować ze względu na prędkość poruszania się, natomiast przykładowy model wykrywający drzwi czy schody, potencjalnie, mógłby nieco poświęcić szybkość detekcji obiektów w ramach zwiększenia dokładności wykrywania .

Celem pracy było stworzenie swoistego generycznego interfejsu, systemu spajającego modele ML w silniku gier, pozwalającego na użycie dowolnego zewnętrznego modelu zależnie od potrzeb jakie mogą przejawiać aplikacje tworzone w Unity, w tym celu poszukiwano potencjalnych rozwiązań oraz dostępnych technologii w tej dziedzinie. Jak się okazuje zakres AI - (ang. Artificial Intelligence) w obszarze Unity wydaje się być świeży, natomiast twórcy Unity już postarali się o kilka patentów dotyczących generyczności w sensie ML.
Jako obiekt doświadczalny, na którego podstawie testowano dostępne metodyki wdrażania i konfigurowania sztucznej inteligencji w Unity opisane w tej pracy, posłużyła przykładowa aplikacja rozszerzonej rzeczywistości przeznaczona do detekcji oraz rozpoznawania obiektów w czasie rzeczywistym. Praca dyplomowa inżynierska rozwijana była równolegle przez p. D. Kobyra: 'Aplikacja mobilna AR z wykorzystaniem natywnych modeli sieci neuronowych w środowisku Unity'

1.2 Zawartość pracy

Zawartość tej pracy jest do pewnego stopnia zbiorem obecnie dostępnych metod wdrażania sztucznej inteligencji do aplikacji stworzonych w silniku Unity3d.

Rozdział 2 Współczesne metody, technologie i koncepcje pozwalające na integrowanie zewnętrznych modeli ML w Unity

Rozdział 3 Mechanizmy ML domyślnie osadzone w silniku Unity

Rozdział 4 Przykładowe problemy inżynierskie oraz aplikacje urzeczywistniające ich rozwiązania przy użyciu silników inferencyjnych, opis trendów związanych z dziedziną ML w Unity

Rozdział 5.1 Wykorzystanie silnika inferencyjnego Barracuda w aplikacji rozszerzonej rzeczywistości przeznaczonej do detekcji obiektów, budowanie aplikacji na mobilne systemy operacyjne

Rozdział 6.2 Podsumowanie pracy, perspektywy dalszego rozwoju

Rozdział 2

Zewnętrzne modele ML w Unity

Jedną z możliwości silnika gier Unity jest integrowanie zewnętrznych modeli ML (ang. Machine Learning'owych). Zewnętrzne modele ML w Unity to wszystkie wytrenowane sieci neuronowe zdefiniowane w jednym z powszechnie używanych framework'ów ML, przykładowo TensorFlow lub Pytorch. Przeznaczenie zewnętrznych modeli ML niekoniecznie musi dotyczyć ich aplikowania w środowisku silnika gry Unity. Przykładowymi środowiskami zastosowań zewnętrznych modeli ML mogą być:

- Systemy samochodów autonomicznych
- Systemy przetwarzania języka naturalnego
- Robotyka i sztuczna inteligencji

W tym rozdziale opisane są obecnie dostępne metodyki, pozwalające na integrowanie takich modeli ML w środowisku Unity, w celu tworzenia aplikacji opartych na technologii sztucznej inteligencji.

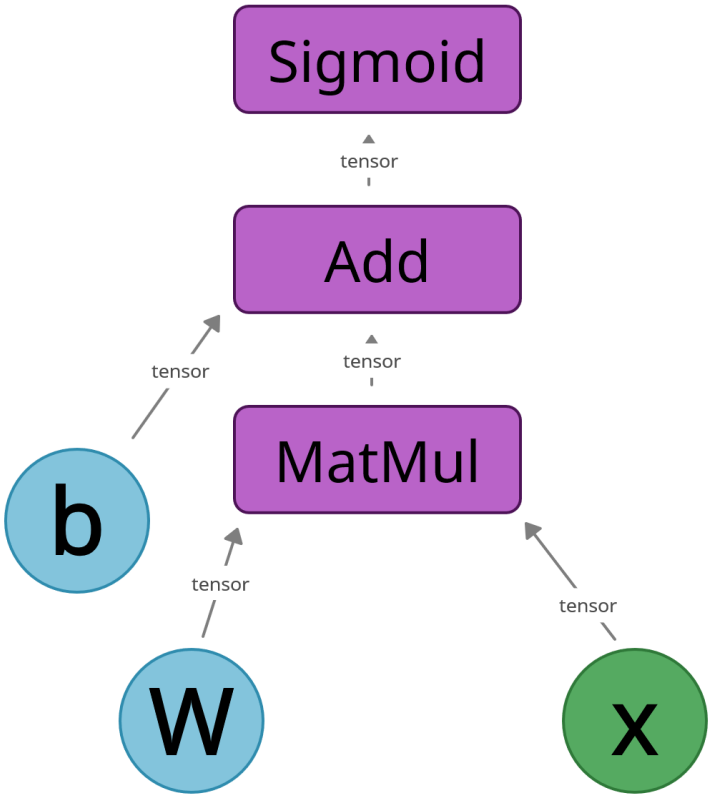
2.1 TensorFlow

TensorFlow (TF) - otwarto źródłowa biblioteka programistyczna napisana przez Google Brain Team w języku C/C++. Wykorzystywana jest w uczeniu maszynowym i głębokich sieci neuronowych. Dostęp do natywnej wersji TF w dowolnym projekcie C/C++ można uzyskać poprzez standardowe linkowanie dynamiczne bibliotek.

Tensorflow jest interfejsem służącym do wyrażania algorytmów uczenia maszynowego, dającym możliwość ich wdrażania oraz wykonywania.

Obliczenia numeryczne w TF są reprezentowane poprzez graf obliczeniowy, który jest podstawą każdego programu napisanego w tej bibliotece, gdzie węzły grafu są traktowane jako operacje numeryczne , a krawędzie grafu są tensorami przepływającymi pomiędzy tymi węzłami.

$$h = Sigmoid(Wx + b)$$

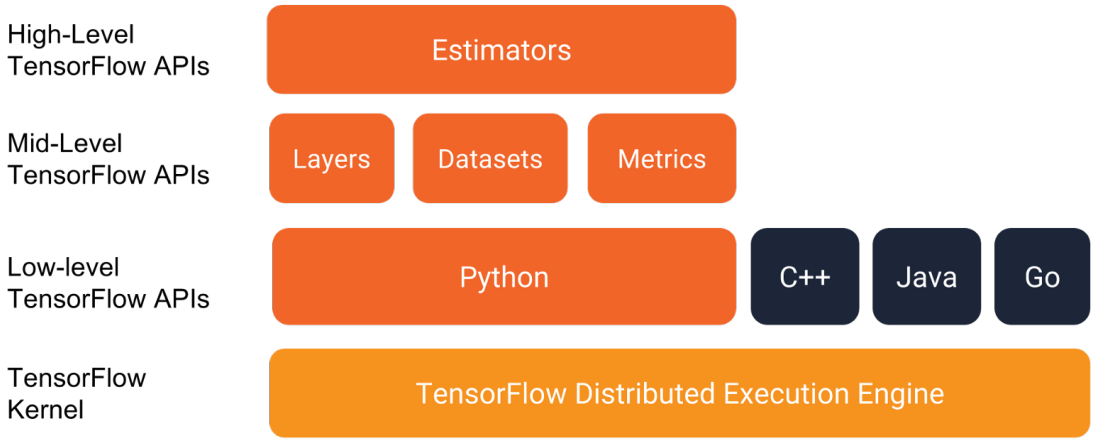


Rysunek 2.1: Przykładowa warstwa sieci neuronowej w rozumieniu grafu TF

W powyższym przykładzie prostej warstwy sieci neuronowej w postaci grafu obliczeniowego znajdują się węzły:

- Węzły stanowe, b i W - zmienne które zwracają ich bieżącą wartość. Poprzez stanowość rozumie się że węzły te zachowują swoją obecną wartość podczas wielu operacji egzekucji warstwy w której się znajdują, taka cecha ułatwia sposób zapisywania i przywracania danych z dysku w dowolnym momencie trenowania modelu. Z definicji grafu obliczeniowego, węzły stanowe również są traktowane jako operacje numeryczne.
- Wejścia (ang. Placeholders), x - są węzłami których wartości są dostarczane w czasie trenowania modelu, traktowane jako standardowe wejście do dowolnej warstwy sieci neuronowej.
- Operacje matematyczne (MatMul: matrix multiplication, Add: dodawanie elementów, Sigmoid: funkcja aktywacji)

Natywna biblioteka TF jest podstawą na której zbudowane są złącza językowe (ang. language bindings) między innymi w Pythonie i C++ jako warstwa wyższego poziomu, te opakowania językowe pozwalają na wygodny sposób odwoływania się do ich natywnej wersji.



Rysunek 2.2: Hierarchia API w TensorFlow

Pythonowa wersja API jest uważana za najwszechstronniejszą z obecnie dostępnych powłok.
W celu dokonania obliczeń zdefiniowanych przez graf obliczeniowy, potrzebny jest obiekt klasy 'Session' który alokuje zasoby a następnie przechowuje faktyczne wartości pośrednich wyników oraz zmiennych. Objekt typu 'Session' hermetyzuje graf w

środowisku w którym wykonywane są obliczenia zdefiniowane przez węzły, na podstawie wartości przekazanych w tensorach.

```
import numpy as np
import tensorflow as tf

# Definicja grafu :

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform(784, 100)),
-1, 1))

x = tf.placeholder(tf.float23, (100, 784))
h = tf.nn.sigmoid(tf.matmul(x, W) + b)

# Uruchomienie grafu w sesji :

sess = tf.Session()
sess.run(tf.initialize_all_variables())
sess.run(h, {x: np.random.random(100, 784)})
```

Listing 2.1: Graf napisany w języku Python na podstawie Rysunku 2.1

Grafy obliczeniowe w TF są tak naprawdę strukturą danych składających się ze zbioru obiektów definiujących operacje numeryczne oraz ze zbioru obiektów typu wielowymiarowych tablic, czyli tensorów. Dlatego że grafy te są strukturą danych, mogą być one zapisywane, uruchamiane, i przywracane, bez posiadania oryginalnego, zazwyczaj pythonowego kodu implementującego taki graf. Grafy zdefiniowane w technologi Tensorflow na skutek swojej elastyczności, mogą być używane w środowiskach które nie posiadają interpretera języka python, np. na serwerze back-end, i również w środowisku silnika Unity.

Zapisywanie wytrenowanego modelu nazywane jest zamrażaniem grafu, a finalnym produktem takiej czynności jest jeden plik o formacie Google .pb

W celu zamrożenia dowolnego grafu, należy załadować go w sesji która posiada już docelowo nastrojone wagi poprzez wcześniejsze trenowanie, ostatecznie używając funkcji biblioteki przeznaczonej do zapisywania grafu

```
output_graph="/graph-model.pb"
with tf.gfile.GFile(output_graph, "wb") as f:
    f.write(output_graph_def.SerializeToString())
sess.close()
```

Listing 2.2: Instrukcja zamrażania grafu TF

W terminologii biblioteki Tensorflow, mylące może być pojęcie modelu, dlatego że modelem ML może być nazwany algorytm uczący sieć neuronową na podstawie danych treningowych, ponadto taką samą funkcję może pełnić graf obliczeniowy, zatem pojęcie grafu TF a pojęcie modelu, zdarza się być wymieniane. Pojęcie modelu również może określać część grafu, sam graf jest zbiorem węzłów i krawędzi które wykonują ustalone zadanie, zatem model może być traktowany jako podzbiór grafu, a graf może składać się z więcej niż jednego modelu. Z punktu widzenia tej pracy, model rozumiany jest jako wytrenowany, wyeksportowany do pliku graf.

2.2 TensorFlowSharp

TensorFlowSharp (TF#) - to złącze językowe .NET do natywnej biblioteki Tensorflow, słowo 'Sharp' pochodzi od C#, standardowego języka framework'u silnika Unity.

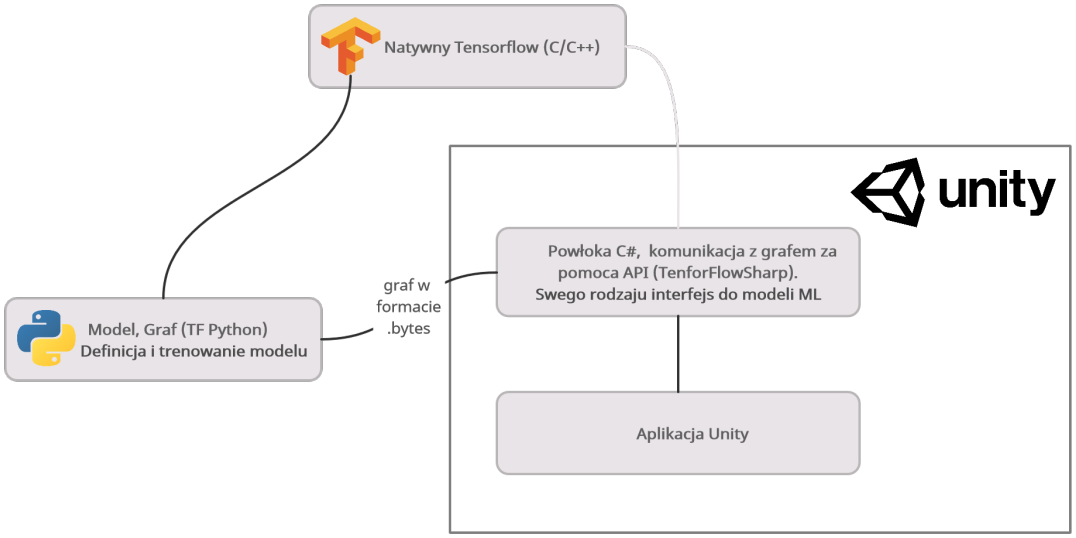
TensorFlowSharp jest nieco wyżej w hierarchii API (Rysunek 2.2), niż wersja napisana w języku Python, a sens użyteczności biblioteki TF# bliższy jest funkcjonalności budowania hipotetycznych, docelowych dla użytkownika aplikacji które konsumują wytrenowane modele w TF, niż samemu definiowaniu grafów.

TensorFlowSharp oferuje możliwość użycia wytrenowanych wcześniej grafów TF, w silniku gry Unity, przebieg tworzenia i integracji takich modeli w Unity odbywa się w trzech kluczowych krokach:

1. Zamrożenie grafu TF w celu użycia go w środowisku Unity
2. Konfiguracja ML-Agends oraz TenserFlowSharp w projekcie Unity
3. Napisanie skryptu C# przy użyciu TF#, który odpowiadałby za przekazywanie i przejmowanie danych z aplikacji Unity do grafu w czasie wykonania programu

Użycie TensorFlowSharp w Unity zależne jest od pakietu ML-Agents, opisanego w (Rozdział 3), który definiuje przestrzeń nazw 'TensorFlow', mimo braku konieczności używania ML-Agents, konieczna jest jego instalacji w przypadku używania TF#

W celu poprawnego zaimportowania zamrożonego grafu TF do projektu Unity, graf musi być w formacie .bytes. W celu konwersji grafu z domyślnego formatu pliku .pb, do formatu .bytes, mogą być wykorzystane instrukcje w (Listing 2.3)



Rysunek 2.3: Szkic prototypu interfejsu do grafów TF w Unity

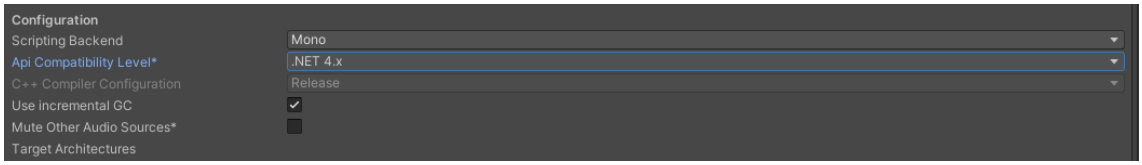
W celu instalacji TF# w środowisku Unity, w pierwszej kolejności, do docelowego projektu należy dodać pakiet ML-Agents. Następnie konieczne będzie pobranie i zainstalowanie plug-in'u TensorFlowSharp. Taki plug-in można pobrać z repozytorium projektu TensorFlowSharp, po pobraniu plug-in'u, wystarczy otworzyć plik przez program Unity, lub przeciągnąć go do folderu 'Assets' projektu.

TenserFlowSharp jest technologią eksperymentalną, dlatego w przypadku używania TF# przez Unity, stąd wymaga konfiguracja do wersji eksperymentalnej:

Wpisanie 'ENABLE_TENSORFLOW' w polu **File –> Build Setting –> Player Settings –> Other Settings –> Configration –> Scripting Define Symbols**

Rysunek 2.4: Uaktywnianie Tensorflow w Unity

Wybranie eksperymentalnej wersji kompatybilnego API (.NET 4.x), w polu **File –> Build Setting –> Player Settings –> Other Settings –> Configration –> Api Compatibility Level**



Rysunek 2.5: Wybieranie kompatybilnego API z Tensorflow

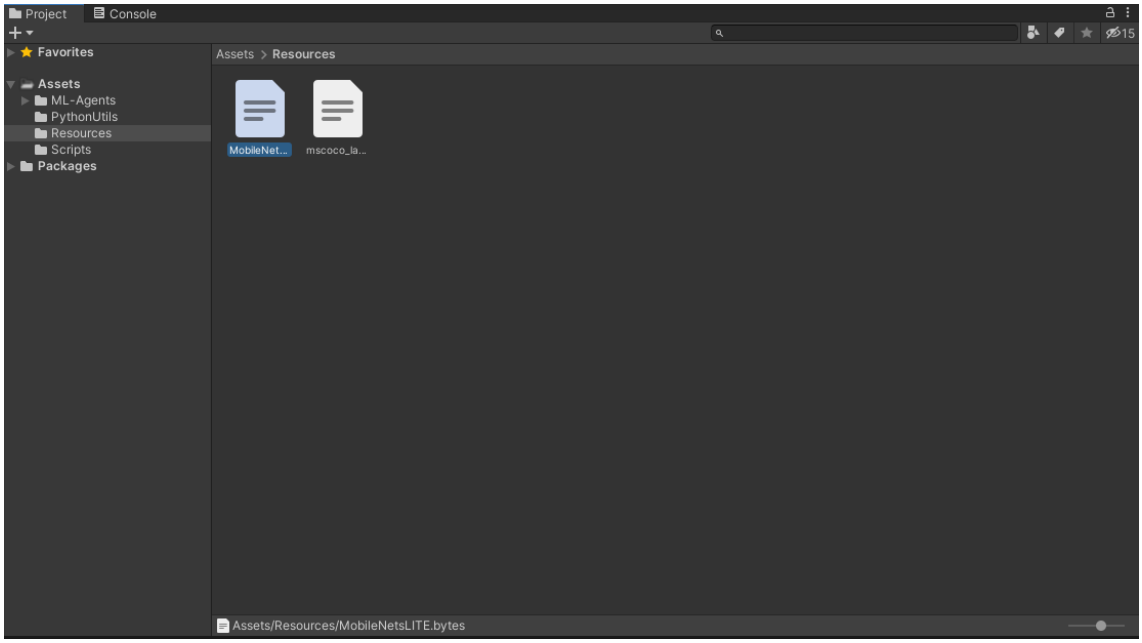
Po wytrenowaniu i zamrożeniu dowolnego grafu TF do pliku .pb, gwoli poprawnego zaimportowania go do środowiska Unity, potrzebne będzie przekonwertowanie zapisanego modelu w formacie .pb do formatu .bytes za pomocą skryptu TF

```
from tensorflow.python.tools import freeze_graph

freeze_graph.freeze_graph(input_graph=model_path+'/graf-wejsciowy.pb',
                input_binary = True,
                input_checkpoint = last_checkpoint,
                output_node_names = "action",
                output_graph = model_path +'/graf-wyjsciowy.bytes' ,
                clear_device =True, initializer_nodes="",input_saver="",
                restore_op_name = "save/restore_all",
    filename_tensor_name = "save/Const:0")
```

Listing 2.3: Instrukcja konwersji grafu z formatu .pb do formatu .bytes

Model zapisany w formacie .bytes wystarczy zaimportować do do folderu 'Assets' znajdującym się w docelowym projekcie środowiska Unity, lokalizacja nie ma większego znaczenia.



Rysunek 2.6: Importowanie grafu do folderu 'Assets' w Unity

Po zaimportowaniu zamrożonego grafu oraz skonfigurowaniu TensorFlowSharp w Unity, korzystanie z API TF# w celu odtwarzania i korzystania z zaimportowanego modelu, powinno przebiegać pomyślnie.

Podstawowe funkcjonalnośći TF# API pozwalające na tworzenie interfejsów dla modeli, grafów w języku C# w środowisku Unity:
API dostępne jest po zaimportowaniu namespace 'Tensorflow'

Jeśli docelowa aplikacja będzie budowana na systemy Android, na poczatku kodu odpowiadającego za interfejs modelu, potrzeba dodać poniższy kod

```
usign Tensorflow;

// dodac do kodu, w przypadku aplikacji docelowej na android
#if UNITY_ANDROID
TensorFlowSharp.Android.NativeBinding.Init();
#endif
```

Listing 2.4: Importowanie przestrzeni nazw Tensorflow oraz konfiguracja aplikacji pod systemy Android

Odtwarzanie zaimportowanego grafu w formacie .bytes jako TextAsset

```
public TextAsset graphModel;

graph = new TFGraph (); // Tworzenie grafu
graph.Import (graphModel.bytes); // Importowanie grafu
session = new TFSession (graph); // Uruchamianie grafu w sesji
```

Listing 2.5: Odtwarzanie zaimportowanego grafu

Nowy objet 'graph' importujący nowy graf, jako wejscie przyjmuje dane w formacie tensora, poniżej przykład z jednowymiarowym tensorem o rozmiarze równym 2

```
var runner = session.GetRunner ();
runner.AddInput (graph ["input_placeholder_name"] [0], new float[]{
    placeholder_value1, placeholder_value2 });
```

Listing 2.6: Instrukcja podawania danych wejściowych do grafu przy użyciu TF#

Należy podać wszystkie wejścia grafu, węzły wejściowe (placeholders), każdy węzeł wejściowy odpowiada wejściu w metodzie 'AddInput'

Pobieranie wyjście grafu po prze-procesowaniu wejścia przez jego warstwy. Poniżej wyjściem grafu jest dwu wymiarowy tensor wartości zmiennoprzecinkowych.

```
runner.Fetch (graph["output_placeholder_name"][0]);
float[,] recurrent_tensor = runner.Run () [0].GetValue () as float[,];
```

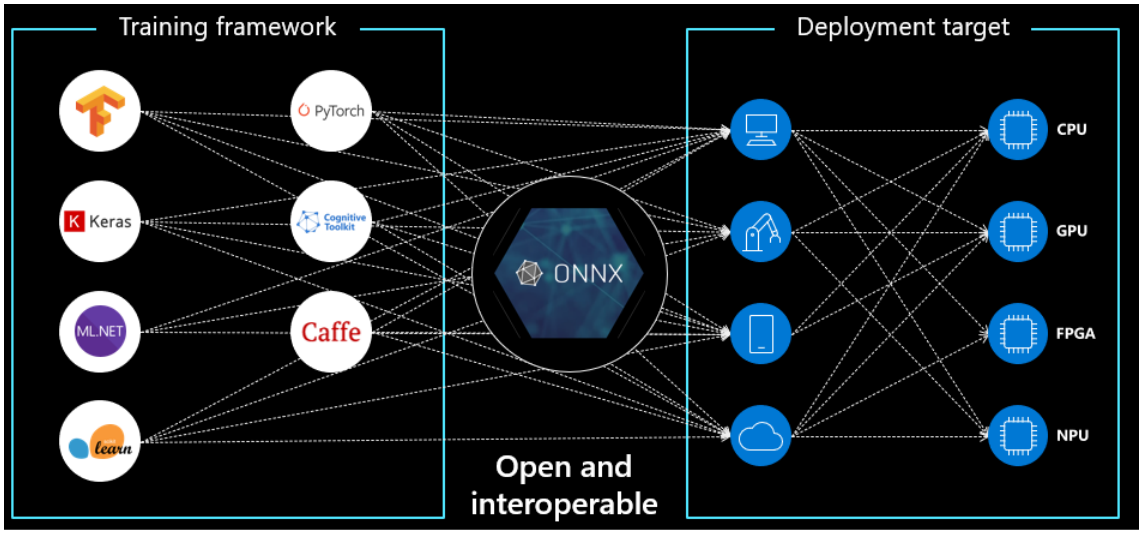
Listing 2.7: Instrukcja fetchowanie wyjścia grafu za pomocą TF#

2.3 Open Neural Network Exchange

ONNX (Open Neural Network Exchange) - to otwarty format stworzony do reprezentowania modeli uczenia maszynowego. ONNX definiuje wspólny zbiór operatorów - elementów składowych modeli uczenia maszynowego, oraz wspólny format pliku w celu umożliwienia używania modeli pochodzących z różnych frameworków, narzędzi, i kompilatorów.

Osoby pracujące nad narzędziami AI, często są ograniczone przez framework lub ekosystem w którym wykonują swoją prace, ONNX jest jedną z pierwszych technologii umożliwiających wymianę modeli ML pomiędzy najpowszechniej używanymi frameworkami.

ONNX umożliwia używanie preferowanego framework'u uczenia maszynowego wraz z dowolnie wybranym silnikiem inferencyjnym.



Rysunek 2.7: ONNX jako punkt zbieżności framework'ów ML | Źródło: https://onnx.ai/

ONNX zaopatruje developerów AI przez rozszerzalny model grafu obliczeniowego, jaki i przez wbudowane operatory oraz standardowe typy danych, w praktyce, ONNX jest otwartym formatem pliku '.onnx' pozwalającym na wymianę modeli pomiędzy różnymi frameworkami oraz narzędziami ML, takimi jak np. Tensorflow, Keras czy Pytorch. Zatem graf zdefiniowany w dowolnym z wymienionych frameworków, może być przekonwertowany do formatu .onnx a następnie odtworzony w dowolnym z pozostałych narzędzi.

Na stronie internetowej `https://onnx.ai/` można dowiedzieć się że ONNX rozwijane jest przez Microsoft, Facebook, będąc równocześnie projektem społecznościowym, zachęcającym do pomocy w rozwoju tej technologii, oraz dzielenia się swoimi pomysłami.

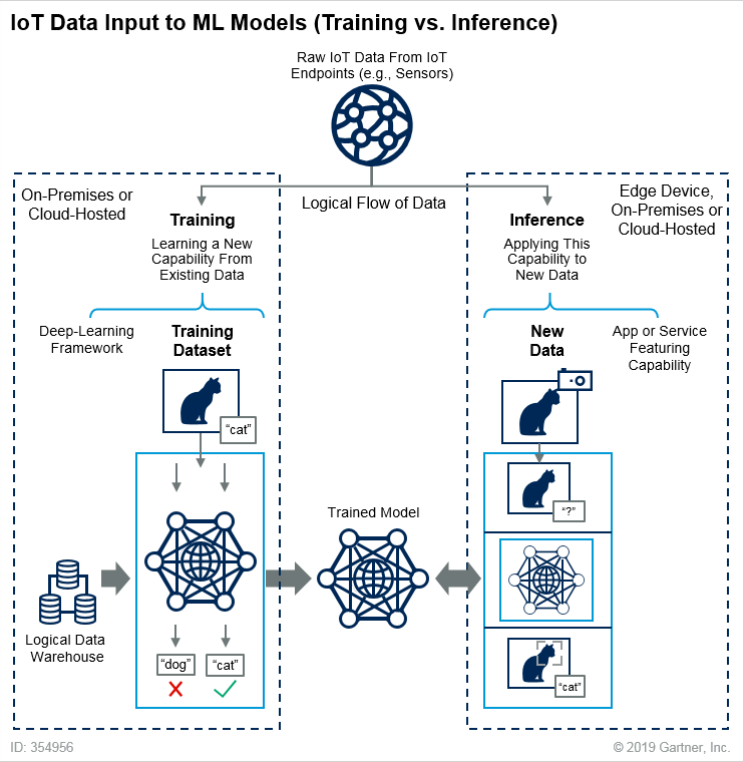
2.4 Biblioteka inferencyjna - Unity Barracuda

Silnik inferencyjny (ang. AI inference engine) - nazywany też serwerem inferencyjnym, to narzędzie służące do wykonywania zadania obsługi AI (AI serving). Jest odpowiedzialny za wdrożenie modelu oraz ewentualne monitorowanie jego wydajności.

Inferencja - jest procesem transferowania danych w czasie rzeczywistym do algorytmów uczenia maszynowego (lub 'modeli ML') w celu obliczenia wyjścia modelu, takiego jak na przykład pojedynczy wynik liczbowy. Taki proces również jest

określany jako 'operacjonalizacja modelu ML' lub 'wprowadzenie modelu ML do produkcji.' Model ML na etapie bycia operacyjnym, jest powszechnie określany mianem sztucznej inteligencji, dlatego że pełni funkcje podobne do myślenia i analizowania przez ludzi.

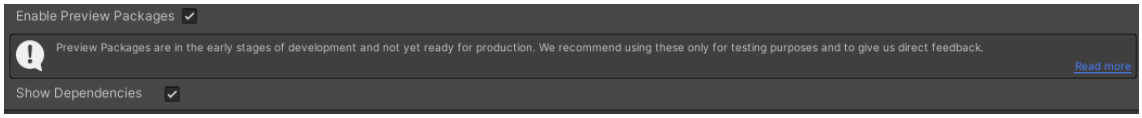
Proces inferencji modelu ML jest bardzo podobny do jego trenowania, z tą różnicą że trenowanie pociąga za sobą konieczność używania frameworku uczenia maszynowego, oraz danych treningowych, inferencja natomiast odbywa się na danych rzeczywistych, przy użyciu serwera inferencyjnego (znanego także jako silnik inferencyjny) który odpowiada za wdrażanie oraz monitorowanie wydajności modelu



Rysunek 2.8: Proces trenowania a inferencja modelu ML | Źródło: https://blogs.gartner.com/paul-debeasi/2019/02/14/training-versus-inference/

Unity Barracuda - jest małą, wielo-platformową developer-friendly biblioteką inferencyjną sieci neuronowych osadzoną w Unity. Barracuda składa się z silnika inferencyjnego dla modeli ML dostępnych w formacie ONNX.

Barracuda dostępna jest poprzez manager pakietów w Unity, **Window -> Package Manager**. W tej lokalizacji sprawdzić flagę **Enable Preview Packages** w **Advance Settings**

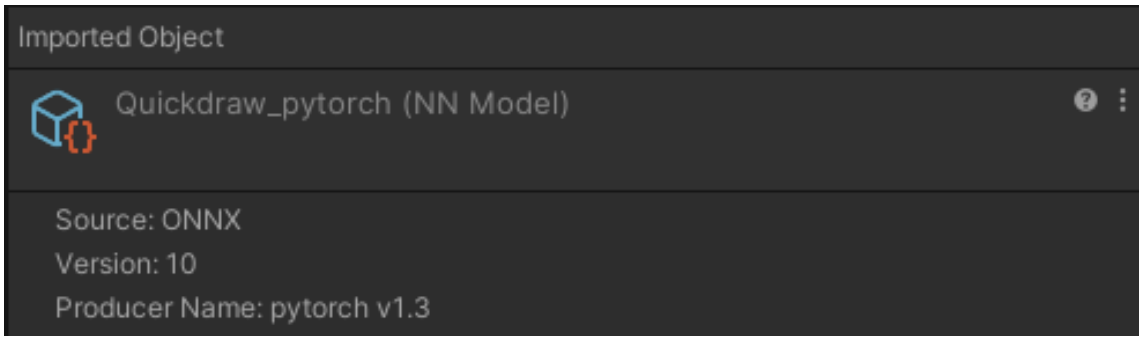


Rysunek 2.9: Aktywowanie podglądu do przed premierowych pakietów Unity

W celu użycia Barracud'y w aplikacji Unity, zwykle potrzebne jest wykonanie poniższych kroków:

1. Zaimportowanie pliku .onnx do docelowego projektu Unity jako zwykły 'Asset'
2. Załadowanie modelu z folderu projektu 'Assets'
3. Stworzenie silnika inferencyjnego (obiekt 'worker')
4. Wprowadzenie modelu ML do produkcji

Dodany plik .onnx do docelowego projektu traktowany jest jako 'asset' typu NNModel



Rysunek 2.10: Przykład 'asset'u' typu NNModel - modelu napisanego w frameworku Pytorch

Pole publiczne typu NNModel pozwala na odwołanie się do powiązanego 'asset'u' przez edytor UI. Klasa 'ModelLoader' pozwala na załadowanie modelu

```
1 public NNModel modelSource;
2 <..>
3 var model = ModelLoader.Load(modelSource);
```

Listing 2.8: Ładowanie modelu onnx przy użyciu Unity Barracuda

Interfejs dostępu do silnika Barracuda nazwany jest jako 'IWorker'. IWorker dzieli model na wykonywalne zadania a następnie rozplanowuje ich realizacje na GPU lub CPU.

Klasa 'WorkerFactory' pozwala na stworzenie, obiektu typu 'Worker', czyli silnika inferencyjnego dla modeli ONNX.

```
Model model = ...

// GPU
var worker = WorkerFactory.CreateWorker(WorkerFactory.Type.
    ComputePrecompiled, model)
var worker = WorkerFactory.CreateWorker(WorkerFactory.Type.Compute,
    model)
// wolne - GPU
var worker = WorkerFactory.CreateWorker(WorkerFactory.Type.ComputeRef,
     model)

// CPU
var worker = WorkerFactory.CreateWorker(WorkerFactory.Type.CSharpBurst
    , model)
var worker = WorkerFactory.CreateWorker(WorkerFactory.Type.CSharp,
    model)
// bardzo wolne - CPU
var worker = WorkerFactory.CreateWorker(WorkerFactory.Type.CSharpRef,
    model
```

Listing 2.9: Tworzenie silnika inferencyjnego dla zaimportowanego modelu ONNX

Po uprzednim załadowaniu modelu oraz stworzeniu obiekt typu 'Worker' odpowiadającego za inferencje modelu, egzekucja modelu odbywa się za pomocą metody 'Execute' która przyjmuje dane wejściowe do modelu. W przypadku gdy objekt ma jedno wejście, jeden węzeł typu 'Placeholder', wejściem będzie jeden Tensor, w przeciwnym razie każde z wejść odpowiadać będzie elementowi słownika przechowującego Tensory

```
    Tensor input = new Tensor(batch, height, width, channels);
    worker.Execute(input);
```

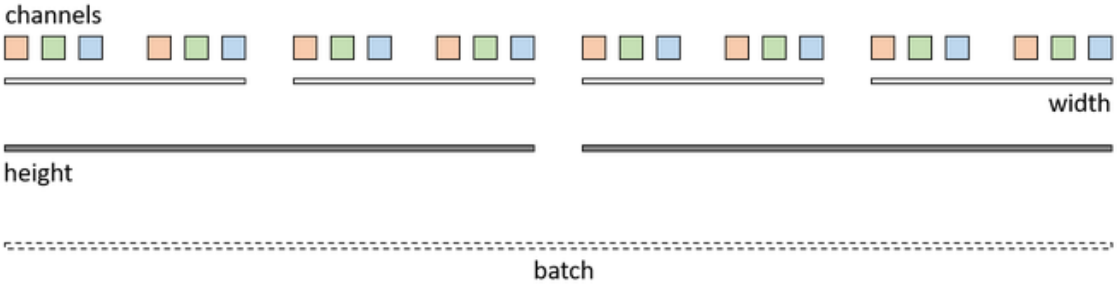
Listing 2.10: Egzekucja modelu ONNX z jednym węzłem wejściowym

```
var inputs = new Dictionary<string, Tensor>();
    inputs[name1] = new Tensor(batch, height1, width1, channels1);
    inputs[name2] = new Tensor(batch, height2, width2, channels2);
    worker.Execute(inputs);
```

Listing 2.11: Egzekucja modelu ONNX z wieloma węzłami wejściowymi

Tensory w bibliotece Barracuda są zaimplementowanie w formie NHCW:

- N - number of samples (batch)
- H - height
- C - channels
- W - height



Rysunek 2.11: Rozkład NHCW formatu Tensorów w bibliotece Barracuda

Natywną kompozycją Tensorów w ONNX jest format NCHW, 'channels-firs'. Barracuda automatycznie konwertuje modele ONNX do formatu NHWC.

Tworzenie Tensorów w Barracuda odbywa się poprzez konstruktor klasy Tensor

```
/* Zwykly batch trzy wymiarowych danych, rozmiar to: batchCount x {
    height, width, channelCount} */
tensor = new Tensor(batchCount, height, width, channelCount);

// batch 1 wymiarowych danych, rozmiar to: batchCount x {elementCount}
tensor = new Tensor(batchCount, elementCount);
```

Listing 2.12: Podstawy konstruowania Tensorów w Unity Barracuda

Barracuda oferuje wiele innych, bardziej zaawansowanych technik konstrukcji, i interakcji z tensorami za pomocą klasy 'Tensor'. Istnieje konstruktor tensorów bezpośrednio z powszechnie używanych w Unity objektów typów Texture2D, Texture2DArray, Texture3D, lub RenderTextra, bez potrzeby dostawania się do indywidualnych pixeli.

```
/* ilosc kanalow odpowiada teksturze 1 - (odcienie szarosci ), 3 - (
    kolor) or 4 - (kolor z kanalem alfa) */
var channelCount = 3;
var tensor = new Tensor(texture, channelCount);
```

Listing 2.13: Konsturowanie tensora bezpośrednio z tekstury Unity

W celu 'fetchowania' wyjścia modelu, po poprzednim podaniu danych wejściowych, używana jest metoda PeekOutput() silnika inferencyjnego, obiekt 'Worker'.

```
var output = worker.PeekOutput(outputName); /* wyjscie o nazwie '
    outputName',  zaleznej od wezla wyjsciowego */
// lub
var output = worker.PeekOutput(); /* zwraca tablice wezlow wyjsciowych
    , w przypadku jednego wezla odwolanie output[0] da zwrocona
    wartosc  */
```

Listing 2.14: Fetchowanie wyjścia modelu ONNX

Ponad funcjonalność fetchowania danych modelu, pochodzących od węzłów wyjściowych, Barracuda umożliwia pobieranie danych z węzłów pośrednich analizowanego modelu. W celu umożliwienia dostępu do pośrednich warst modelu przez metode fetchującą wyjście, w konstruktorze worker'a podaje się ich nazwy, takie jak zdefiniowano w modelu ONNX.

```
// lista posrednich wezlow
var additionalOutputs = new string[] {"layer0", "layer1"}
m_Worker = WorkerFactory.CreateWorker(<WorkerFactory.Type>,
    m_RuntimeModel, additionalOutputs);
...
var outputLayer0 = worker.PeekOutput("layer0");
var outputLayer1 = worker.PeekOutput("layer1");
/* mozliwosc zapytan o wezly wyjsciowe wciaz jest dostepna */
var output = worker.PeekOutput(outputName);
```

Listing 2.15: Fetchowanie wyjścia oraz warst pośrednich modelu ONNX

Metoda PeekOutput(), działa w sposób redukujący ilość alokacji pamięci, zatem nie kopiuje aktualnej wartości tensora pochodzącego z wyjścia modelu. Tak więc ponowne wywołanie modelu, zwróci tensor z nowymi danymi a poprzedni zostanie nadpisany. Do wykonania kopii tensora, służy metoda worker.Fetch(), należy jednak pamiętać, że każdy ze skopiowanych tensorów trzeba ostatecznie zwolnić z pamięci komputera, w tym celu służy metoda tensor.Dispose().

Unity Barracuda oferuje wiele innych możliwości składających się na bardzo elastyczną i wygodną biblioteką inferencyjną. Barracuda jest obecnie nową, wciąż rozwijającą się technologią. Aktualnie wspieranymi architekturami sieci neuronowych, oraz modelami są:

- Wszystkie modele ML-Agents
- MobileNet v1/v2 - klasyfikator obrazu
- Tiny YOLO v2 - detektor objektów
- Modele U-Net
- Konwolucyjne sieci neuronowe
- Gęste sieci neuronowe (Dense neutral neworks)

2.5 Barracuda a TensorFlowSharp

Wspólną cechą technologii TensorFlowSharp oraz Unity Barracuda, jest to że obie są bibliotekami bezpośrednio dostępnymi w środowisku Unity. TF# to złącze językowe .NET do natywnej biblioteki Tensorflownie i nie jest docelowo stworzone z myślą o Unity, niemniej jednak umożliwia tworzenie w nim swego rodzaju interfejsów dla modeli stworzonych w frameworku Tensorflow. Barracuda w przeciwieństwie do TF, jest rozwijana przez Unity-Technologies, i jest małą, wieloplatformową developer-friendly biblioteką inferencyjną sieci neuronowych osadzoną w Unity, silnik inferencyjny Barracud'y pozwala na korzystanie z modeli ONNX, a zatem modeli z powszechnie używanych frameworków, nie tylko Tensorflow.

Rozdział 3

Wewnętrzne mechanizmy ML w Unity

3.1 Unity ML Agents

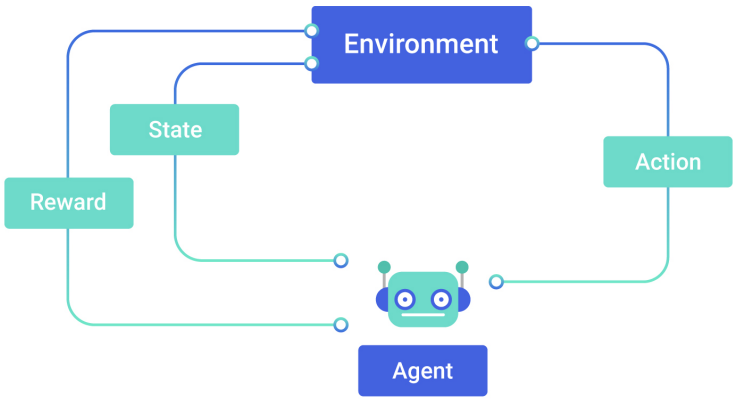
Unity Machine Learning Agents Toolkit (ML-Agents) - jest swego rodzaju predefiniowanym szkieletem AI, dokładniej, to otwarto źródłowy projekt który pozwala projektom stworzonym w silniku Unity, służyć jako środowisko do trenowania inteligentnych agentów. Agenci mogą być uczeni poprzez metodyki takie jak:

- Uczenie przez wzmacnianie (ang. Reinforcement learning)
- Uczenie przez naśladowanie (ang. Imitation learning)
- Neroewolucja (ang. Neuroevoluton)
- Inne metody uczenia maszynowego dostępne poprzez proste w użyciu Python API

Agenci mogą zostać wykorzystani do wielu celów, w tym do kontrolowania zachowań NPC lub automatyzacji testowania oraz ewaluowania różnych decyzji projektowych gier przed ich wydaniem.

W środowisku Unity, agentem może być dowolny GameObject (dowolna postać w scenie projektu), agent generuje obserwacje, dane wejściowe do modeli ML, oraz wykonuje akcje który otrzymuje jako odpowiedź. Każdy agent jest połączony z zachowaniem.

Zachowanie agenta definiuje konkretne atrybuty, takie jak ilość akcji które może wykonać. Zachowanie można traktować jako funkcję która otrzymuje obserwacje od agenta, a zwraca akcje. Te zachowania dzieli się na trzy rodzaje, nauczone (learning), heurystyczne (heuristic) lub inferencyjne (inference)



Rysunek 3.1: Data flow diagram w przypadku metody uczenia przez wzmacnianie

- Zachowanie nauczone - jest to zachowanie które nie jest zdefiniowane, ale nauczone przez samego agenta
- Zachowanie heurystyczne - jest zdefiniowane poprzez mocno zakodowany zbiór zachowań
- Zachowanie inferencyjne - to takie które zawiera plik sieci neuronowej, po uczeniu sieci zachowania, zachowanie staje się zachowaniem inferencyjnym

ML-Agents zapewniają wszystkie potrzebne narzędzia do używania Unity jako silnika symulacyjnego służącego do uczenia tak zwanych 'polityk', polityką są zazwyczaj sieci neuronowe.

Zestaw narzędzi ML-Agents składa się z pięciu wysokopoziomowych komponentów:

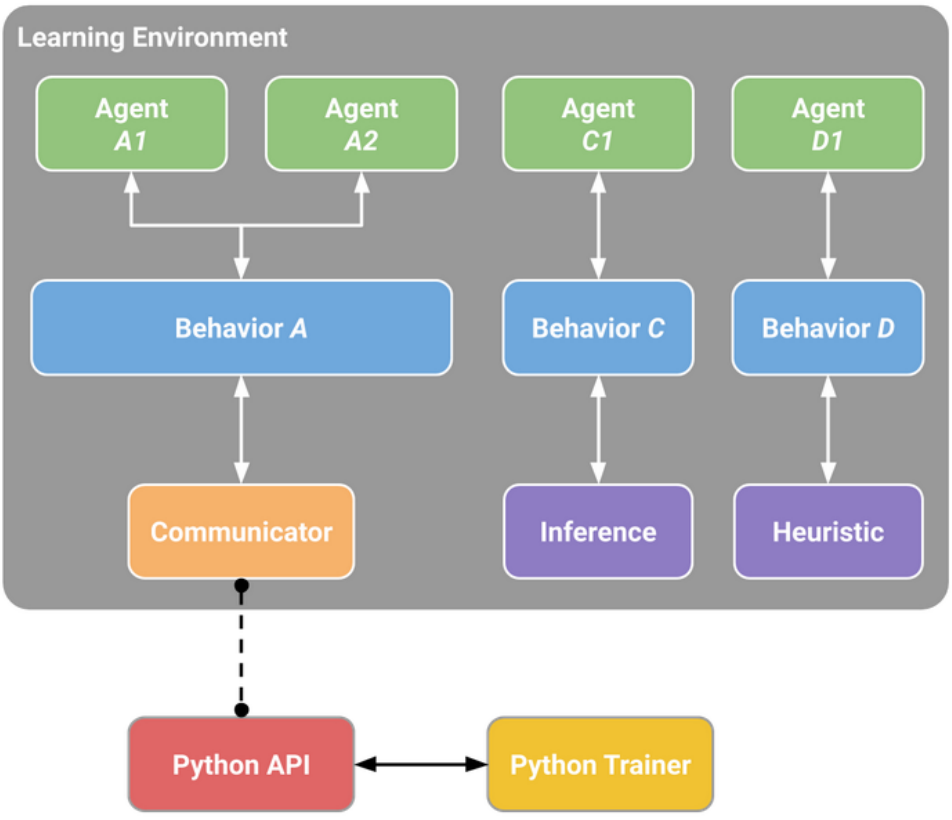
- Środowisko nauczania (ang. Learning Environment) - zawiera scenę Unity oraz wszystkie postacie gry. Scena Unity zapewnia środowisko w którym agent obserwuje, działa, oraz uczy się.
- Nisko poziomowe Python API - zawiera nisko poziomowy interfejs w języku Python stworzony do interakcji oraz sterowania ze środowiskiem nauczania.

Python API nie jest częścią Unity, znajduje się poza jego środowiskiem i komunikuje się poprzez zewnętrzny komunikator. API zawiera się w dedykowanym pakiecie Python 'mlagents_envs' dostępnym poprzez manager pakietów 'pip'

- Zewnętrzny komunikator - łączy środowisko nauczania z nisko poziomowym Python API. Komunikator znajduje się w środowisku nauczania

- Algorytmy trenujące (ang. Python Trainers) - zawierają wszystkie algorytmy nauczania maszynowego które pozwalają na uczenie agentów. Te algorytmy są zaimplementowane w języku Python oraz są częścią pakietu 'mlagents'. Pakiet 'mlagents' udostępnia komendę wiersza poleceń 'mlagents-learn' która mieści wszystkie metody trenowania oraz opisane opcje.

(Rysunek 3.2) przedstawia strukturę środowiska ML-Agents, zbudowaną na podstawie wymienionych powyżej komponentach



Rysunek 3.2: Diagram zestawu narzędzi ML-Agents | Źródło: `https://github.com/Unity-Technologies/ml-agents/`

ML-Agents dostarcza kilka implementacji najnowocześniejszych algorytmów do trenowania inteligentnych agentów. Mówiąc dokładniej, podczas treningu. każdy agent w scenie Unity, wysyła swoje obserwacje do Python API poprzez zewnętrzny komunikator. Python API procesuje otrzymane obserwacje a następnie odsyła akcje do wykonania przez każdego z agentów. Podczas trenowania tych akcji, Python API dobiera najlepszą politykę, zazwyczaj sieć neuronową, dla każdego z agentów. Gdy faza trenowania zakańcza się, polityka dla każdego z agentów może być wyeksportowana jako plik z modelem. Następnie w fazie inferencji, każdy z agentów wciąż wysyła swoje obserwacje, lecz zamiast wysyłać je do Python API, obserwacje przesyłane są do wbudowanego w agenta modelu.

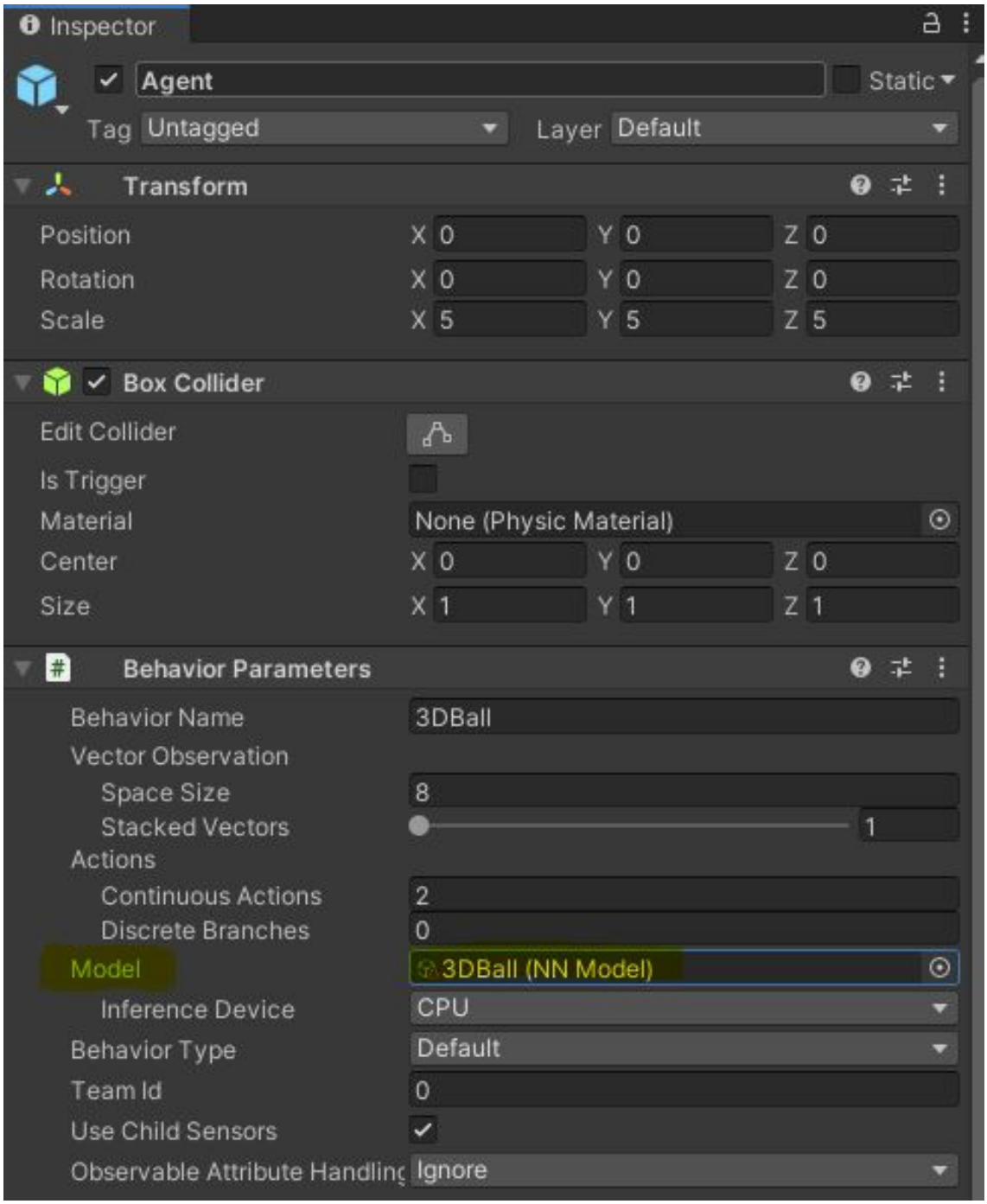
3.2 Unity ML Agents a Barracuda

ML-Agents podobnie jak Barracuda dostepne jest poprzez manager pakietów w Unity. Ważnym aspektem jest to że ML-Agents polega na silniku inferencyjnym Unity Barracuda, w celu uruchamiania modelu w scenie Unity, w taki sposób aby agenci mogli podejmować optymalne akcje w każdym z kroków inferencji. Dodatkowo każdy z użytkowników ML-Agents może użyć swoich algorytmów nauczania, w takim przypadku wszystkie zachowania agentów będą kontrolowane z środowiska Python.

Zatem ML-Agents jest niejako fuzją technologii TensorFlowSharp oraz Barracuda. Zewnętrzny komunikator pomiędzy środowiskiem Unity a Python API jest zdefiniowany w TF#, natomiast Barracuda pozwala zestawowi narzędzi ML-Agents na używanie pre-trenowanych sieci neuronowych w grach Unity.

Barracuda domyślnie, poza środowiskiem ML-Agents, pozwala na używanie modeli .onnx składających się z coraz szerszego zestawu architektur uczenia maszynowego. Natomiast w przypadku inferencji modeli dla ML-Agents, modele te muszą być dostosowane bezpośrednio do środowiska Unity, to znaczy że zewnętrzne modele, te nie znajdujące się domyślnie w pakiecie języka Python - 'mlagents', z którego korzysta ML-Agents, są zmuszone przestrzegać pewnych przyjętych konwencji nazw węzłów oraz tensorów, co ogranicza tworzenie innych architektur modeli docelowych dla ML-Agents, niż te odgórnie zdefiniowane w pakiecie 'mlagents'

W celu użycia modelu w ML-Agents, wystarczy w inspektorze docelowego agenta przeciągnąć plik modelu, do pola 'Model'.



Rysunek 3.3: Inspektor przykładowego agenta sceny Unity

Rozdział 4

Silniki Inferencyjne a problemy inżynierskie

Silnik inferencyjny (ang. AI inference engine) - nazywany też serwerem inferencyjnym, to narzędzie służące do wykonywania zadania obsługi AI (AI serving). Jest odpowiedzialny za wdrożenie modelu oraz ewentualne monitorowanie jego wydajności.

Pojawiają się przypadki, w których sztuczna inteligencja zastępuje stażystów, wykonujących schematyczne prace, jak na przykład klasyfikowanie opinii klientów na temat komercyjnej aplikacji. Obecnie do oceny dowolnego tekstu można wykorzystać odpowiedni system NLP (ang. Natural Language Processing), opierający się na metodyce analizy sentymentu. Takie rozwiązanie zwróciłoby nacechowanie emocjonalne i wydźwięk opinii autora , natomiast w przypadków gdy potrzeba segregacji na większej ilości etykiet, sprawa nieco się komplikuje. Dla przykładu, opinie na temat aplikacji mogłyby być pogrupowane jako te związane z UI, z jej wydajnością, lub inne, nie do końca związane z samą aplikacją, a w celu automatycznej kwalifikacji tekstu na dowolnie przyjęte etykiety, istnieje duże prawdopodobieństwo że potrzebny będzie model ML uszyty na miarę. W celu doboru oraz użycia modelu zależnie od potrzeby etykietowania analizowanego tekstu, posłużyć może system zawierający silnik inferencyjny.

4.1 Przykładowe problemy i rozwiązania

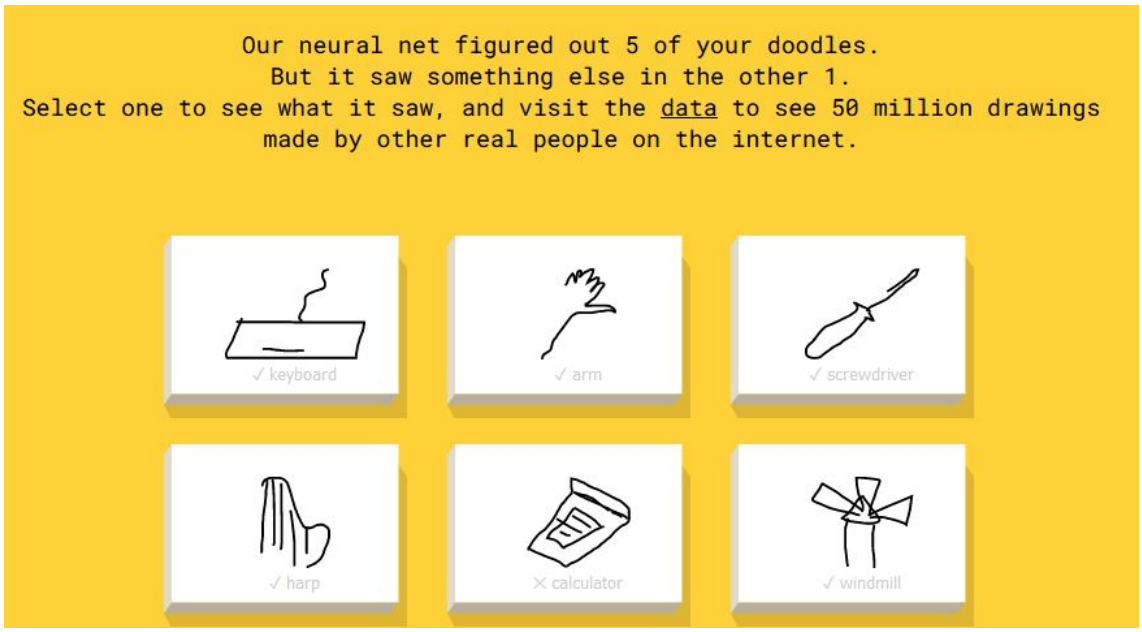
W tej sekcji znajduje się krótka prezentacja dwóch przykładowych projektów Unity, wykorzystujących modele ML w jego środowisku, za pomocą silnika inferencyjnego

zawartego w bibliotece Barracuda.

4.1.1 Klasyfikacja obrazu

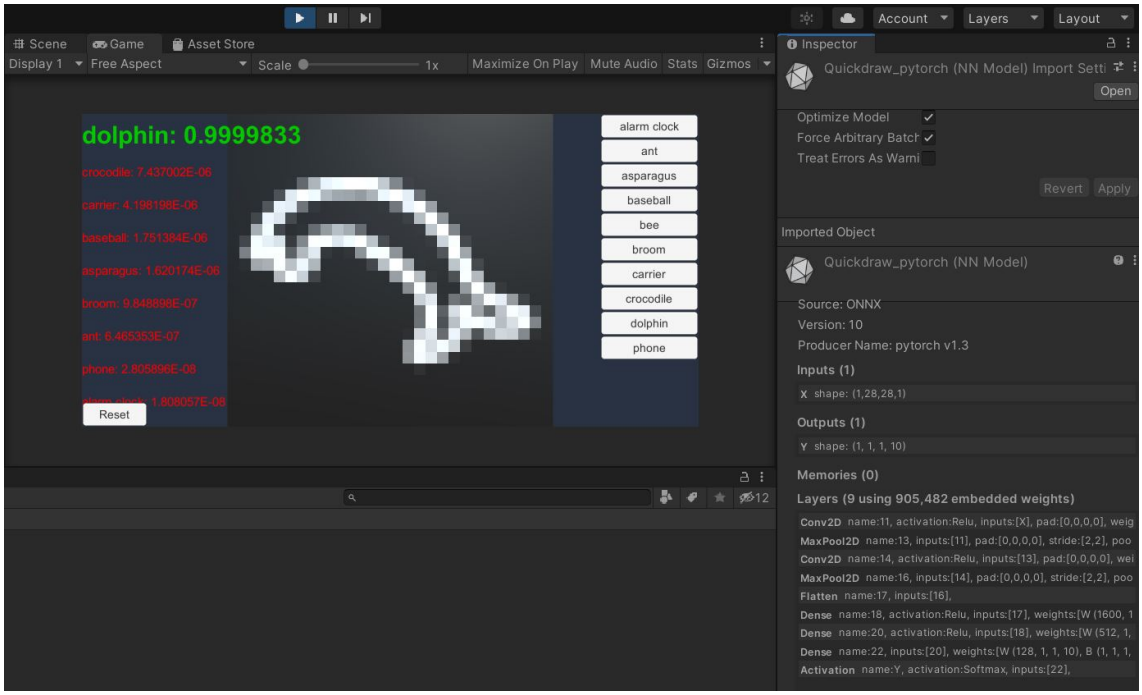
Wraz z dostępem do zewnętrznych modeli ML w środowisku Unity, wykrywanie, rozpoznawanie, oraz klasyfikowanie obiektów w scenie Unity nie sprawia większego problemu, wystarczy dowolny model pełniący podobne funkcje, oraz zależny od potrzeb skrypt C# definiujący zachowanie gry lub symulacji.

AI-Drawing - dostepny pod: [3] - to przykładowy projekt Unity stworzony na wzór aplikacji 'Quick Draw!'. 'Quick Draw!' to mini gra online opracowana przez Google, która wzywa graczy do narysowania obiektu lub pomysłu, a nastepnie używa sztucznej sieci neuronowej do odgadnięcia, co przestawia rysunek. AI uczy się na podstawie każdego nowego rysunku, zwiększając tym swoją zdolność do klasyfikowania obiektów w przyszłości.



Rysunek 4.1: Przykład wyniku gry w 'Quick Draw!'

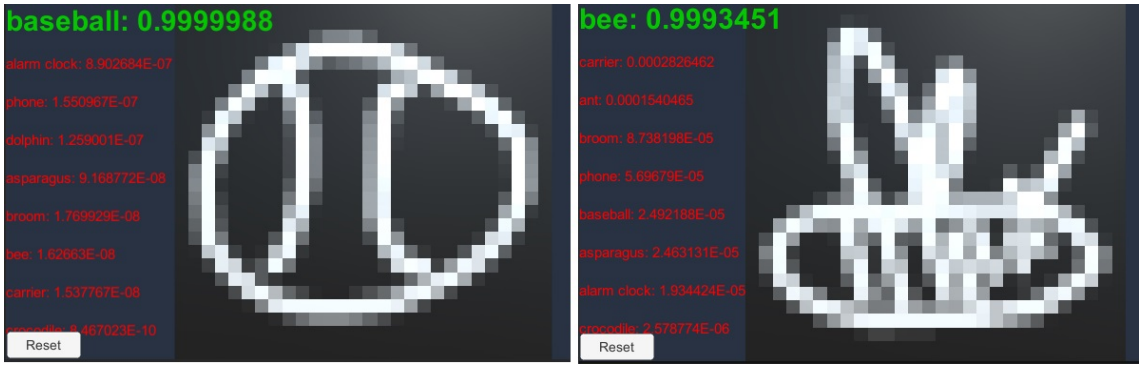
Model używany przez Quick Draw! rozpoznaje około 300 klas. AI-Drawing korzysta z 9 warstwowego modelu zdefiniowanego w frameworku Pytoch, który nauczony jest rozpoznawać 10 klas, na bazie danych wykorzystanej do trenowania modelu aplikacji Quick Draw!.



Rysunek 4.2: Widok projektu AI-Draving oraz inspektor modelu NN

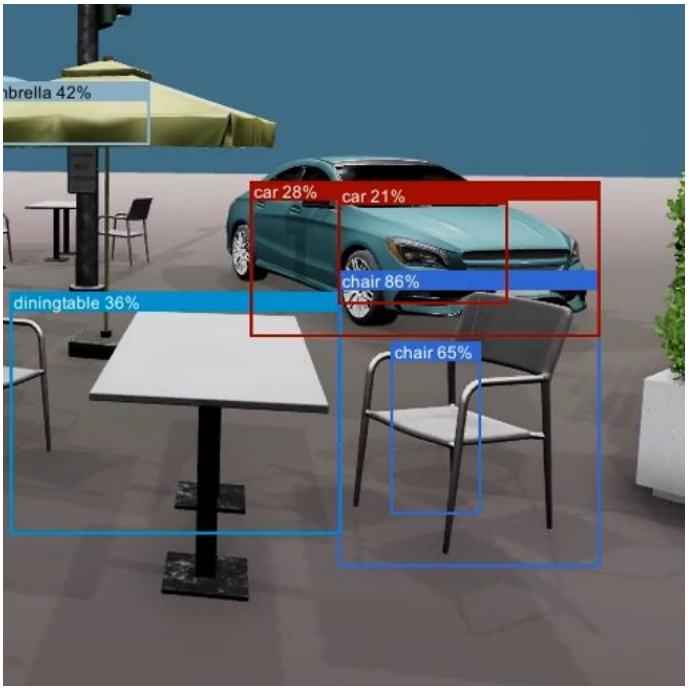
W inspektorze modelu używanego przez projekt AI-Drawing, okno po prawej stronie na powyższym rysunku, widać wszystkie warsty modelu, nazwy jego węzłów wejściowych oraz wyjściowych, wraz z wymiarami tensorów - X: shape(1, 28, 28, 1) to wymiar tensora wejściowego, Y: shape(1, 1, 1, 10) to wymiar tensora wyjściowego, model składa się miedzy innymi z dwóch warstw konwolucyjnych (Conv2D), najczęściej stosowanych podczas analizy obrazów wizualnych, funkcja aktywacji to popularny Softmax.

AI-Drawing składa się z dwóch krótkich skryptów, Painter.cs, obsługujący funkcjonalność rysowania na kanwasie - element UI Unity, oraz BarracudaTest.cs, części definiującej metody odpowiadające za inferencje z modelem na podstawie tekstury 2D pobranej z kanwasu. Po lewej stronie od kanwasu, znajduje się lista wyników otrzymanych z modelu, wartości pewności wystąpienia obiektu na obrazie, pochodzącego z kanwasu. Wartość najbliższa jedynce, czyli największe prawdopodobieństwo, odpowiada finalnemu wynikowi przypuszczenia klasyfikacji obiektu.



Rysunek 4.3: Wynik działania aplikacji AI-Drawing

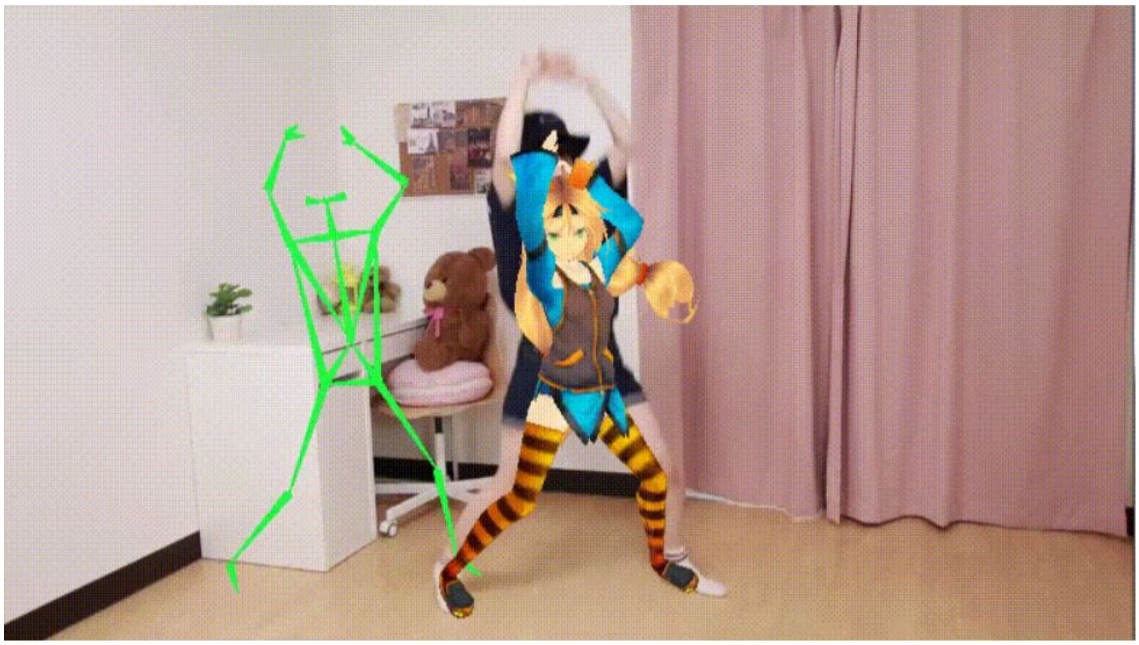
Dla modeli ML, źródło pochodzenia danych wejściowych nie ma większego znaczenia. W przypadku obrazu, źródłem może być obiektyw kamery, lub wirtualna scena w grze komputerowej, która zazwyczaj przypomina świat realny. Zatem modele przeznaczone do detekcji, klasyfikacji obiektów, będą działać porównywalnie w środowisku wirtualnym jak i w świecie namacalnym. Dzięki tej sposobności, symulacje wirtualne mogą posłużyć jako środowiska do generowania danych treningowych w szybki, przystępny i elastyczny sposób. Poniżej przykład działania modelu w Unity, opartego na systemie YOLO służącego do wykrywania obiektów .



Rysunek 4.4: Wynik działania modelu YOLO w Unity

4.1.2 Przechwytywanie ruchu

ThreeDPoseUnityBarracuda - dostepny pod: [11] - to kolejny przykładowy projektu Unity wykorzystujący bibliotekę Barracuda. Projekt ten korzysta z modelu ML, standardowo dostepnym w pliku .onnx, który pozwala na estymowanie pozy w 3D (ang. 3D pose estimation) w Unity. ThreeDPoseUnityBarracuda pobiera obraz z kamery w czasie rzeczywistym który wysyłany jest jako wejście do modelu ML. Jeśli na obrazie zostanie wykryta jedna osoba, na jej miejsce nałożony zostanie gotowy awatar, naśladujący ruchy widocznej osoby.



Rysunek 4.5: Wizualizacja działania projektu ThreeDPoseUnityBarracuda

O aplikowaniu technologi przechwytywaniu ruchu, słyszy się najczęściej w branży gier oraz filmów animowanych, w tych specjalnościach najczęściej wykorzystuje się kostium do nagrywania ruchów, dzięki niemu zarejestrowane postacie poruszają się bardzo naturalnie i realistycznie. Kostiumy do nagrywania ruchów są bardzo drogie i zazwyczaj dostępne dla większych komercyjnych twórców, dodatkowo przechwytywanie ruchu ma wiele innych aplikacji w których estymacja pozy odbywająca się przy użyciu kamery jest przyzwoitsza niż przy użyciu specjalnego do tego sprzętu. Przechwytywanie ruchu oraz estymacja pozy wykorzystywana jest w medycynie, w celu wykrywania anomalii w posturze pacjentów, kolejnym zastosowaniem jest tłumaczenie języka migowego oraz wykrywanie pieszych przez samochody autonomiczne.

Estymacja pozy z obrazu 2D jest wymagającym zadaniem, dlatego ze ludzie mają różnorodne wyglądy, kamera będąca źródłem obrazu, może padać pod różnymi kontami, dodatkowo kończyny człowieka bywają czasami zasłonięte przez resztę ciała, w takim przypadku modele ML estymują ich pozycje na podstawie widocznej części osoby. Biblioteka Barracuda umożliwiła tworzenie aplikacji w świecie wirtualnym, jak i rozszerzonej rzeczywistości, przy wykorzystaniu dostępnych modeli odpowiadający za estymowanie pozy. W następnej sekcji, opisany jest bardzo popularny trend, wykorzystujący technologię estymacji pozy w Unity.

4.2 Trendy ML w Unity

4.2.1 Virtual Youtubers (vTubers)

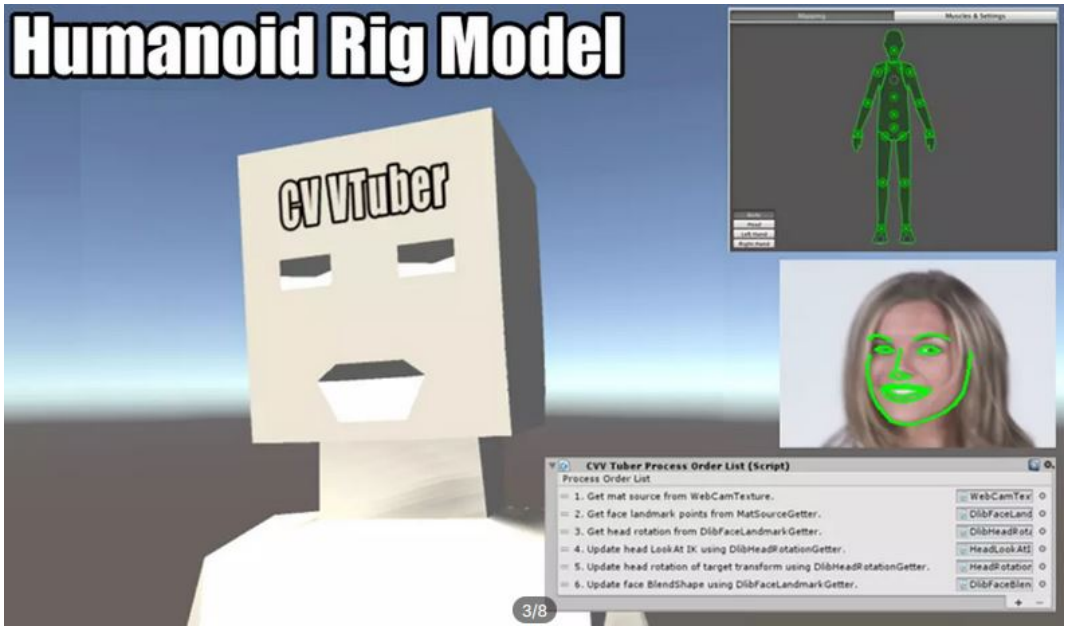
W 2020 roku z powodu kwarantanny, zmuszeni byliśmy do pozostawania w domu, a większość kontaktów międzyludzkich odbywała się zdalnie przez komunikatory online. Brak możliwości rozrywki w świecie rzeczywistym, niejako zachęcił wiele osób do ucieczki w świat wirtualny. 41% samo określających się graczy we Francji, oznajmiło że grało w więcej gier ze względu na pandemię, a branża gier miała się jeszcze lepiej w czasie panowania COVID-19, jako że klienci byli zmuszeni do pozostawania w domu.

W 2020 roku można było dostrzec wyraźny wzrost popularności vTuber'ów (virtual YouTuber) - osób nagrywających filmy, lub prowadzących działalność streaming'owe, używając przy tym cyfrowego awataru wygenerowanego przy użyciu grafiki komputerowej. Ten rosnący trend narodził się w Japonii w połowie 2010 roku, dopiero 10 lat później, vTuberzy stali się modą w krajach zachodnich. Przykładem osoby będącej vTuberem która w 2020 roku zyskała dużą popularność, o średniej ilości widzów przebijających zwykłych twórców, jest 'Codemiko'. Codemiko jest streamer'em na platformie Twitch.tv, jest również doświadczoną osoba w dziedzinie animacji oraz grafiki komputerowej 3D. Codemiko, stworzyła avatar 3D w silniku Unreal Engine, kontrolowany przez kostium do przechwytywania ruchu - Xnens. Proces rozwijania, projektowania awataru 'Miko', pozwolił ostatecznie na mapowanie awataru oraz jego płynne poruszanie się.

Rysunek 4.6: Streamer Codemiko podczas tworzenia awataru 'Miko' w Unreal Engine

Rysunek 4.7: Codemiko podczas jednego ze stream'ów na platformie Twitch.tv

Kostium do przechwytywania ruchu - Xens, może kosztować ponad 30 tyś. dolarów, a mimo istniejących zniżek dla developerów, cena wciąż pozostaje ogromną sumą. W Unity pojawiają się już gotowe, darmowe Assety pozwalające na zostanie vTuberem, opierające swoje działanie na estymacji pozy oraz przechwytywaniu ruchu twarzy. Przykładowy asset to 'CV VTuber', stworzony przez Enox Software.



Rysunek 4.8: Widok jednej ze scen w 'Assecie' Unity: CV-VTuber

Dokładność mapowania awataru do osoby bez specjalnego sprzętu przeznaczonego do przechwytywania ruchu, pozostawia jeszcze wiele do życzenia, a programy służące do transmisji strumieniowej nie mają żadnych opcji dotyczących wirtualnego streamingu. Z pewnością są potencjalni klienci, których ucieszyłaby wiadomość o przyjaznym dla użytkownika oprogramowaniu, służącym do mni. wirtualnego stremowania w przystępnej jakości modeli awatarów, oraz dokładności estymacji pozy.

Rozdział 5

Aplikacja AR oparta o silnik inferencyjny Barracuda

5.1 Interfejs dla modeli YOLOv2-tiny przy uzyciu Barracuda

YOLO (You Only Look Once) - jako algortym do wykrywania obiektów w czasie rzeczywistym, jest obecnie jednym z najefektywniejszych algorytmów tego typu. YOLOv2-tiny to mniejsza odmiana wersji drugiej algorytmu YOLO, który zaczął stawiać pierwsze kroki w 2015 roku. Obecnie modele oparte na architekturze YOLO, do wersji drugiej włącznie, są wspieranie przez silnik inferencyjny Barracuda.

W celu przetestowania poprawności działa biblioteki Barracuda, posłużyła aplikacja AR Unity, przeznaczona na urządzenia mobilne. Funkcja aplikacji polega na wykrywaniu obiektów klas, zależnych od nauczonego modelu YOLOv2-tiny. Analizowana aplikacja pobiera obraz z kamery telefonu w czasie rzeczywistym, używając do tego biblioteki AR-Foundation. Obraz ten trafia następnie do podpiętego w ten czas modelu YOLO dostepnym w pliku .onnx, ostatecznie rysując ramki na obrazie w miejscach znajdowania się ewentualnych, wykrytych przez model obiektów. Poniżej znajduje się metoda 'Detect' aplikacji, która przedstawia moment inferencji aplikacji z modelem. Funkcja ta jako argumenty przyjmuje obraz oraz funkcje wywołania zwrotnego, która jako wejście otrzymuje pozycję wszystkich wykrytych obiektów. Ciałem funkcji 'Detect', są metody biblioteki Barracuda, odpowiadające za tworzenie słownika tensorów, podawanie danych wejściowych do modelu oraz fetchowanie

wyjścia modelu.

```
public IEnumerator Detect(Color32[] picture, System.Action<IList<
BoundingBox>> callback){
    using (var tensor = TransformInput(picture, IMAGE_SIZE,
IMAGE_SIZE))
    {
        var inputs = new Dictionary<string, Tensor>();
        inputs.Add(INPUT_NAME, tensor);
        yield return StartCoroutine(worker.StartManualSchedule(
inputs));
        //uzycie workera - inferencja z modelem ML
        var output = worker.PeekOutput(OUTPUT_NAME);
        var results = ParseOutputs(output);
        var boxes = FilterBoundingBoxes(results, 5,
MINIMUM_CONFIDENCE);
        callback(boxes);
    }
}
```

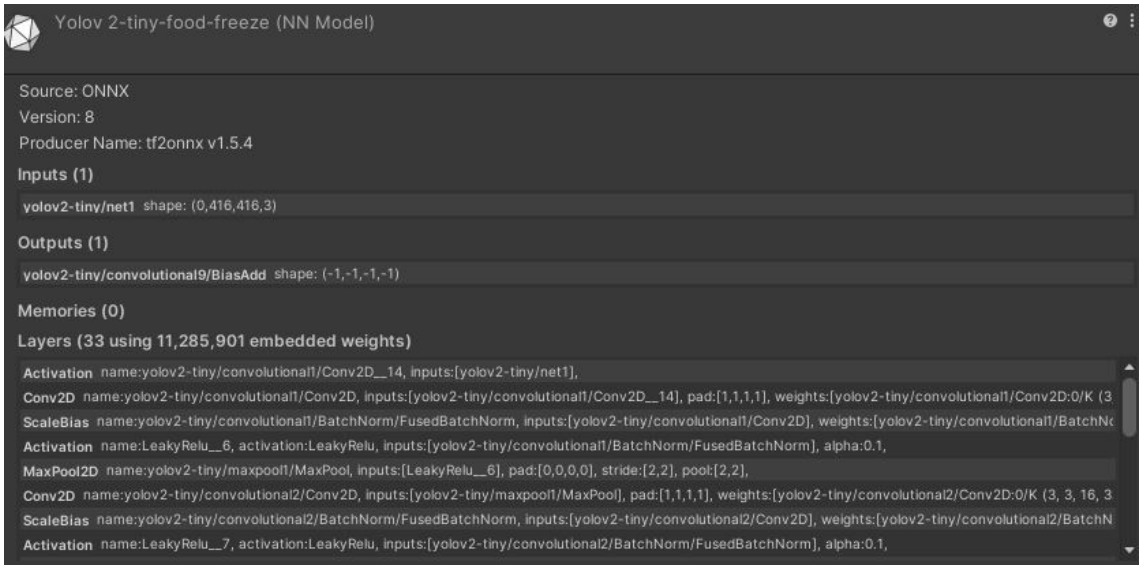
Listing 5.1: Definicja funkcji 'Detect' odpowiadającej za inferencje z modelem ONNX

W ramach testowania wykorzystano dwa modele YOLOv2-tiny:

Model	**Liczba klas**	**Wejście modelu**	**Wyjscie modelu**
yolov2-tiny-food-freeze.onnx	100	yolo2-tiny/net: shape(0, 416, 416, 3)	yolo2-tiny/ convolutional9/BisadAdd: shape(-1, -1, -1, -1)
yolov2-tiny-voc.onnx	20	input_1: shape(-1, 416, 416, 3)	cov2d_9/BiasAdd: shape(-1, -1, -1, -1)

Pierwszy z nich składał się z 33 warst, wejściem tego modelu był węzeł o nazwie 'yolo2-tiny/net' przyjmujący tensor o kształcie (0, 416, 416, 3), wyjściem modelu był węzeł 'yolo2-tiny/convolutional9/BisadAdd', (-1, -1, -1, -1). Model ten nauczony jest na zbiorze danych FOOD100 dataset, który składa się ze zdjęć 100 klas potraw kulinarnych. Poniżej znajduje się widok inspektora Unity pierwszego modelu.

Drugi rozpatrywany modele o architekturze YOLOv2-tiny, posiadał 40 warstw, o wejściu - input_1 shape: (-1, 416, 416, 3) oraz wyjściu - cov2d_9/BiasAdd: shape(-1, -1, -1, -1)



Rysunek 5.1: Inspektor Unity pierwszego testowanego w aplikacji AR modelu YOLOv2-tiny, dostępnego w ONNX

Rysunek 5.2: Wynik działania drugiego modelu w analizowanej aplikacji AR

Rysunek 5.3: Inspektor Unity drugiego testowanego w aplikacji AR modelu YOLOv2-tiny, dostępnego w ONNX

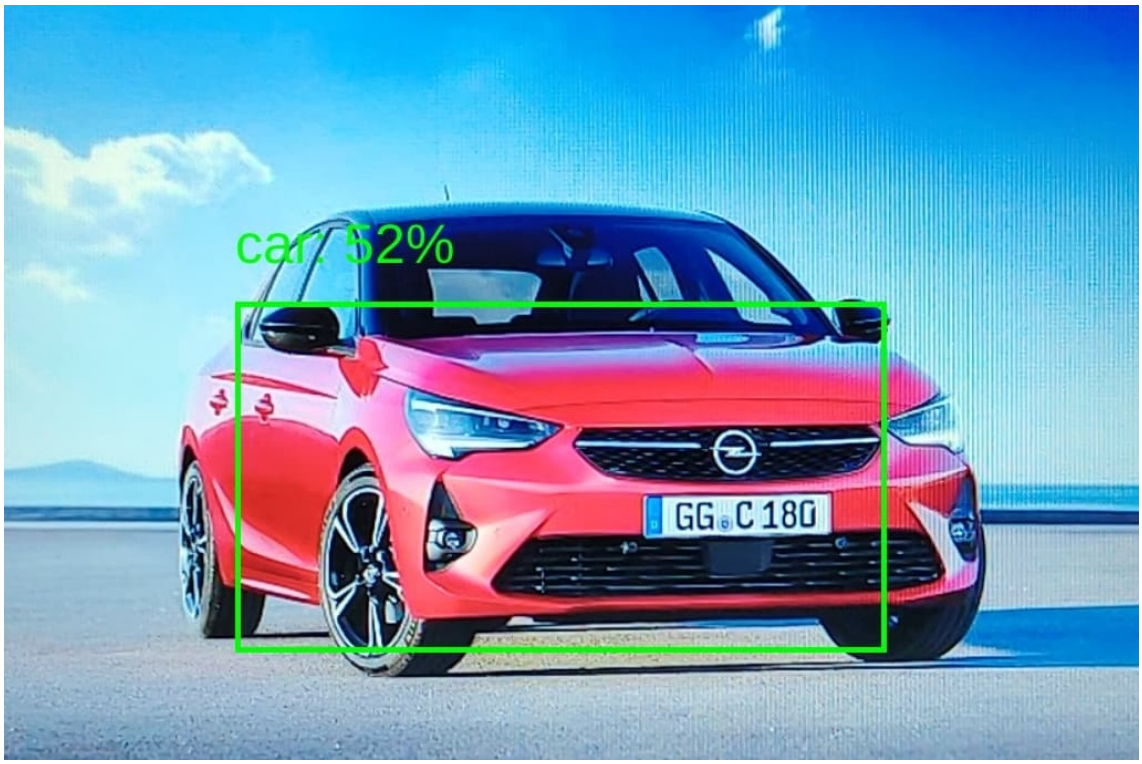
Nazwy wejścia i wyjścia obu analizowanych modeli różnią się. W testowej aplikacji, inferencja z modelem za pomocą silnika Barracuda, klasy 'Worker', odbywa się poprzez podanie nazw węzła wejściowego i wyjściowego modelu. Zatem podczas zmiany modelu, w celu jego popranego działania, wymagana jest zmiana wejść i wyjść w obiekcie 'worker', na te zdefiniowane w modelu. W projekcie aplikacji AR Unity, zmienne te przechowywane były w dwóch prywatnych polach.

```
1    // ONNX model input and output name. Modify when switching models.
2    private const string INPUT_NAME = "input_1";
3    private const string OUTPUT_NAME = "conv2d_9/BiasAdd";
```

Listing 5.2: Nazwy węzłow modelu Barracuda

Ze względu na strukturę kodu projektu, trzeba również podać ilość wykrywanych klas. W przypadku drugiego modelu jest ich 20, składają się one z powszechnie spotykanych obiektów, takich jak pies, samochód, czy osoba.

Podczas użycia drugiego modelu, aplikacja AR również działa poprawnie. Ponadto, wedle oczekiwań, wykrywa inne obiekty, nauczone przez drugi model, niż podczas inferencji z pierwszym modelem. Na (Rysunek 5.4) widać jak aplikacja wykrywa obiekt klasy 'car'.



Rysunek 5.4: Wynik działania pierwszego modelu w analizowanej aplikacji AR

Rozdział 6

Podsumowanie

6.1 Wnioski

Silnik gier Unity, poza możliwością tworzenia zaawansowanych gier oraz aplikacji w technologi rzeczywistości wirtualnych, pozwala również na ich usprawnianie, w ten sam sposób w jaki technologia AI udoskonala świat realny. Ponadto, dzięki metodykce wykorzystywania sztucznej inteligencji w środowisko silników gier, Unity odnajduje swoje zastosowanie, jako laboratorium przeznaczone do trenowania modeli ML oraz generowania danych treningowych, usprawniających nasza rzeczywistość. Liderem wśród obecnych technologi AI w Unity wydaje się być biblioteka inferencyjna Barracuda. Barracuda sprawia wrażenie bycia przyjaznej dla użytkownika, a jej czołową cechą jest elastyczność pochodząca z dostępu do modeli, będących do dyspozycji poprzez format pliku ONNX. Każdy z modeli stworzony w dowolnym, powszechnie używanym frameworku, między innymi TensorFlow, Pytorch, czy Keras, może być przekonwertowany do formatu ONNX. Z kolei technologia TensorFlowSharp, podobnie jak Barracuda, również pozwala na tworzenie interfejsów dla modeli ML, natomiast jedynie dla modeli stworzonych w bibliotece TF, ponadto TF# nie jest technologią przygotowaną z myślą o Unity, zatem używanie TF# w środowisku gier bywa problematyczne, jak i samo budowanie aplikacji korzystających z tej technologii. Zastaw narzędzi ML-Agents jest wartościowy, wspólnie dla twórców gier, jak i developerów systemów AI, jako że zapewnia platformę, na której systemy AI mogą być oszacowywane w bogatym środowisku Unity. Unity poszerza granice możliwości badań naukowych nad sztuczna inteligencją, tym samym, udostępniając modele ML dla społeczności oraz środowiska gier i symulacji komputerowych.

6.2 Perspektywy rozwoju

Efektem końcowym przeglądu dostępnych technologii zawartych w niniejszej pracy jest interfejs dla konkretnych modeli ML (w formacie ONNX), przeznaczonych do detekcji obiektów w docelowej aplikacji AR. Zamiana modeli ML używanych przez tą aplikacje, mogła by być możliwa w czasie wykonywania programu, przy użyciu biblioteki inferencyjnej Barracuda. W celu osiągnięcia takich rezultatów, można byłoby zaimplementować odpowiedni interfejs użytkownika, pozwalający na dobranie pliku .onnx docelowego modelu, oraz odpowiednie pola wejścia tekstowe, pozwalające na zmianę nazw węzłów wejściowych i wyjściowych modelu. Zmiana nazw wejścia i wyjścia używanego przez aplikacje AR modelu, na poprawne dla niego nazwy, mogła by być również procesem automatycznym. Obecnie biblioteka Barracuda wspiera modele detekcji obiektów stworzone w architekturze YOLO, do wersji drugiej włącznie, oraz inne zaprojektowane w formach wymienionych w sekcji (Biblioteka inferencyjna - Unity Barracuda). Jednak może zdarzyć się że Barracuda nie będzie wstanie obsłużyć modelu o niekonwencjonalnej architekturze, zatem monitorowanie zdolności, oraz ciągle postępujący rozwój Barracud'y za pośrednictwem samego Unity Technologies, wciąż jest istotny. Biblioteka Barracuda w dużej mierze polega na możliwościach ekosystemu sztucznej inteligencji ONNX - który jest projektem społecznościowym, zachęcającym do angażowania się w jego rozwój.

Bibliografia

[1] Facebook/Microsoft. Onnx. https://onnx.ai/, 2020.

[2] Hocking, J. *Unity in Action, Second Edition.* Manning Publications, 2018.

[3] Marchlewicz, M. Ai-drawing. https://github.com/MarekMarchlewicz/AI-Drawing, 2020.

[4] migueldeicaza. Tensorflowsharp. https://github.com/migueldeicaza/TensorFlowSharp, 2019.

[5] Trends, Y. C. . Youtube culture trends report: Vtubers.

[6] Unity-Technologies. barracuda-release. https://github.com/Unity-Technologies/barracuda-release, 2019.

[7] Unity-Technologies. Ml-agents. https://github.com/Unity-Technologies/ml-agents, 2019.

[8] Unity-Technologies. Barracuda. https://docs.unity3d.com/Packages/com.unity.barracuda@1.0/manual/index.html, 2020.

[9] Wikipedia. Open Neural Network Exchange — Wikipedia,. http://en.wikipedia.org/w/index.php?title=Open%20Neural%20Network%20Exchange&oldid=994701693, 2021.

[10] Wikipedia. TensorFlow — Wikipedia,. https://pl.wikipedia.org/wiki/TensorFlow, 2021.

[11] yukihiko. Threedposeunitybarracuda. https://github.com/digital-standard/ThreeDPoseUnityBarracuda, 2020.