WPF Verteilte Systeme

A distributed Architecture for training a Deep Learning Agent

Jonas Faßbender

Amjad Haddad

Hamza Aldada

David ...

Contents

List of Figures

1 Introduction

This document records our attempt of building a distributed application for training a Deep Learing Agent to master a specific environment.

It should be noted that the focus of this project lies on "distributed", which means the main goal of this application is speeding up a task which requires lots of computing power using technologies and algorithms for concurrent computing in order to horizontally scale up and reduce the overall time the task takes for completing rather than an attempt to find a optimized algorithm that reduces time complexity of training a Deep Learning Agent.

We will begin with introducing the technologies we used, starting with a summary of the Python programming language we have used for building our application, followed by a brief introduction to Neural Networks. After that we will outline the frameworks and libraries we added in order to build our Machine Learning Infrastructure, before finally presenting the concept of a Message Broker, which is one crucial part of our application's architecture.

After that we will present the architecture and the concepts of our application with emphasis on how we achieve concurrency on different levels. This is followed by a chapter which outlines our test results. The document is concluded by giving an overwiew over improvements and optimizations which can further increase the results of our application.

The whole project can be found at:

https://github.com/jofas/wpfvs

2 Technologies

2.1 Python

Figure 1: Python Logo

At the point when computers came to presence in their initial years, there was the need for them to be modified.

Computer scientists came with a 0 and 1 programming language as the optimum solution at that time, yet it was extremely troublesome and tedious. This trouble caused great advancements in programming and prompted the processes of making high level languages that we know and hear about nowadays.

Python is a general-purpose high-level programming language[24] whose design philosophy emphasizes code readability.[25]

Python aims to combine "remarkable power with very clear syntax"[26] and its standard library is large and comprehensive. Its use of indentation for block delimiters is unusual among popular programming languages.

Python can be considered as a multi-paradigm programming language. Rather than forcing programmers to adopt a particular style of programming, it permits several styles: object-oriented programming and structured programming also are fully supported. Many other paradigms are supported using extensions, such as pyDBC[27] and Contracts for Python[28] which allow Design by Contract.

Python uses dynamic typing and a combination of reference counting and a cycle-detecting garbage collector for memory management. An important feature of Python is dynamic name resolution (late binding), which binds method and variable names during program execution.

Rather than requiring all desired functionality to be built into the language's core, Python was designed to be highly extensible. New built-in modules can be easily written in C, C++. Python can also be used as an extension language for existing modules and applications that need a programmable interface. This design of a small core language with a large standard library andan easily extensible interpreter was intended by Van Rossum from the very start because of his frustrations with ABC (which espoused the opposite mindset).[29]

Like other dynamic languages, Python is often used as a scripting language, but is also used in a wide range of non-scripting contexts. Using third-party tools, such as Py2exe or Pyinstaller[30], Python code can be packaged into standalone executable programs. Python interpreters are available for many operating systems.

2.1.1 History

Python was conceived in the late 1980s[3] and its implementation was getting started in December 1989 [4] by Guido van Rossum at CWI in the Netherlands as a successor to the ABC programming language (itself inspired by SETL)[5] capable of exception handling and interfacing with the Amoeba operating system. [6]

Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, Benevolent Dictator for Life (BDFL).

Python 2.0 was released on 16 October 2000, with many major new features including a full garbage collector and support for Unicode. However, the most important change was to the development process itself, with a shift to a more transparent and community-backed process.

Python 3.0, a major, backwards-incompatible release, was released on 3 rd of December 2008[7] after a long period of testing. Many of its major features have been back-ported to the backwards-compatible Python 2.7.[8]

2.1.2 What is Python?

Python is a high-level programming language designed to be easy to read and simple to implement. It is open source, which means it is free to use, even for commercial applications.

Python can run on Mac, Windows, and UNIX operating systems and has also been ported to Java and .NET virtual machines. [9]

Python is considered a scripting language, like Ruby or Perl and is often used for creating Web applications and dynamic Web content. It is also supported by a number of 2D and 3D imaging programs, enabling users to create custom plug-ins and extensions with Python. Examples of applications that support a Python API include GIMP, Blender, and Autodesk Maya.[9]

Python supports the use of modules and packages, which means that programs can be designed in a modular style and code can be reused across a variety of projects. Once the user develops a module or package she/he needs, it can be scaled for use in other projects, and it's easy to import or export these modules.[10]

Scripts written in Python (.py files) can be parsed and run immediately. They can also be saved as a compiled programs (.pyc files), which are often used as programming modules that can be referenced by other Python programs. [9]

That being said, it can be considered that python has a lot of distinguishing properties and features as being portable since, it can be used on a lot of operating systems. In addition there are some other properties such:

1. Python supports other technologies:

   It can support COM, .Net, etc. objects. Also, some alternatives and complements were created for Python that make it easier to work with these objects in an integrated mode.

2. Presence of Third Party Modules:

   The Python Package Index (PyPI) contains numerous third-party modules that make Python capable of interacting with most of the other languages and platforms.[11]

3. Python is open source:

   Even though all rights of this program are reserved for the Python institute, but it is open source and there is no limitation in using, changing and distributing.[12]

4. Learning Ease and Support Available:

   Python offers excellent readability and uncluttered simple-to-learn syntax which helps beginners to utilize this programming language. The code style guidelines, PEP 8, provide a set of rules to facilitate the formatting of code. Additionally, the wide base of users and active developers has resulted in a rich internet resource bank to encourage development and the continued adoption of the language.[11]

5. Productivity and Speed:

   Python has clean object-oriented design, provides enhanced process control capabilities, and possesses strong integration and text processing capabilities and its own unit testing framework, all of which contribute to the increase in its speed and productivity. Python is considered a viable option for building complex multi-protocol network applications.[11]

2.1.3 Features and philosophy

Python uses dynamic typing and a combination of reference counting and a cycle-detecting garbage collector for memory management. An important feature of Python is dynamic name resolution (late binding), which binds method and variable names during program execution.[14]

The design of Python offers only limited support for functional programming in the Lisp tradition. The language has map, reduce and filter functions, comprehensions for lists, dictionaries, and sets, as well as generator expressions.[34] The standard library has two modules (itertools and functools) that implement functional tools borrowed from Haskell and Standard ML.[15]

An important goal of the Python developers is making Python fun to use. This is reflected in the origin of the name which comes from Monty Python, and in an occasionally playful approach to tutorials and reference materials, for example using spam and eggs instead of the standard foo and bar.[35]

Python has a large standard library, commonly cited as one of Python's greatest strengths, providing tools suited for many tasks. This is deliberate and has been described as a "batteries included" Python philosophy. For Internet-facing applications, a large number of standard formats and protocols (such as MIME and HTTP) are supported. Modules for creating graphical user interfaces, connecting to relational databases, pseudorandom number generators, arithmetic with arbitrary precision decimals, manipulating regular expressions, and doing unit testing are also included.[35]

Some parts of the standard library are covered by specifications (for example, the WSGI implementation wsgiref follows PEP 333), but the majority of the modules are not. They are specified by their code, internal documentation, and test suite (if supplied). However, because most of the standard library is cross-platform Python code, there are only a few modules that must be altered or completely rewritten by alternative implementations.[35]

The standard library is not essential to run Python or embed Python within an application. Blender 2.49 for instance omits most of the standard library.

Figure 2: Popular software written in Python

2.1.4 Why we took Python for our project

Python is a well-designed language that can be used for real world programming. Some of its strengths were very appealing to us when we thought about which language to use.

The most obvious strength and the reason we decided to use Python are the libraries, frameworks and APIs for machine learning written for/in Python (especially Tensorflow and Keras, which we used, but including far more well designed and beloved tools like PyTorch, Theano, matplotlib, etc).

Here are some other strengths of Python we really appreciated during development:

- System programming:

  Internal interfaces of python that are created for working with services of operating system cause Python to be a suitable language for system programming.

  These interfaces provide some functions such as: files and directories operations, parallel processing, etc. the standard library of Python can support the different types of platforms and operating systems. It contains some tools for working with system resources such as: environmental variables, files, sockets, pipes, processes, multiple treats, command line, standard stream interfaces, shell programming, etc. [35]

- Network and internet programming:

  Various modules are embedded in Python standard library that provide many tools for network programmers, such as: client-server connection, socket programming, FTP, Telnet, email functions, RPC, SOAP, etc.

  Also, some third-party tools like mod-Python allow web servers like apache to run Python scripts. Furthermore, some popular programs such as: Django, Turbo gears, Pylon, Zope and Web Ware support Python scripts.[35]

- Numerical programming:

  Python with the NumPy library for Linear Algebra computations (used in Tensorflow) can be a powerful alternative for FORTRAN and C++, because it provides powerful tools for working with mathematical libraries, by using simple Python code. Also, there are many third-party tools on the internet for numerical computations.

2.2 Neural Nets

2.2.1 Derivation from biology

"Neural networks are an attempt to model the insights gained in brain research on the interaction between nerve cells (neurons) and connections (synapses)".[37]

Here, an artificial neuron mimics the functioning of a nerve cell . A biological nerve cell is connected to other nerve cells via synapses. At these connections, stimuli are transmitted by means of chemical messengers (neurotransmitters) and converted into an electrical signal within the nerve cell.

The synapses of a nerve cell are located at so-called dendrites (branches), from which the transmitted stimuli are transmitted to the soma (cell body). It does matter how far the synapse is from the soma. The closer a synapse is to the soma, the stronger the stimulus transmission.[33, 37]

In addition, it should be the case that multiple synapses stimuli at the same time which adds incoming stimuli within the nerve cell.[33] If the stimulus transmitted exceeds a threshold within the cell, then an action potential is triggered and the cell forwards the signal to other cells, which in turn are connected via synapses to the stimulating transmitting cell. Nerve cells form associations in the human brain, which develop the ability to recognize complex structures. Artificial neural networks are an attempt to imitate these associations of nerve cells.[37]

2.2.2 Structure of an artificial neuron

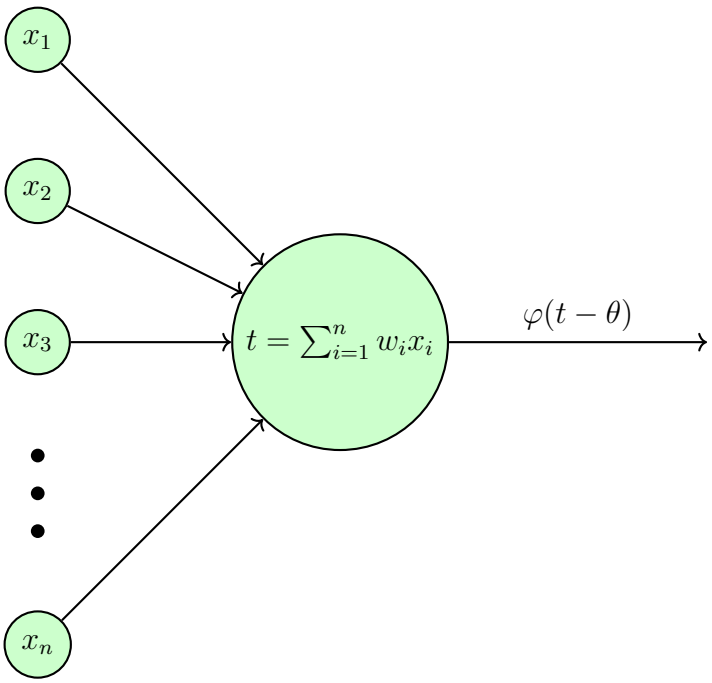

Figure 3: Scheme of an artificial neuron

From chapter 2.2.1, the components and functioning of an artificial neuron can be derived.

In an artificial neuron the following happens:

1. An artificial neuron is connected to other neurons or the direct value input via input connections, which are supposed to represent the synapses of the nerve cell (described in Figure 3 with x_i). These inputs can be discreet or continuous.[36]

If the data comes from other neurons, the weighting of the connection is additionally transferred to the value ω_i . Each connection of a neuron has an individual weighting. The greater the value of this weighting, the more important the transmitted value x_i for the network output.

2. The value and weight of each input connection is added by the transfer function $\sum_{i=1}^{n} w_i x_i$ to calculate the net input value t.

3. Every artificial neuron, like a nerve cell, has a threshold. An artificial neuron is a value θ. The threshold θ is subtracted from the net input value t to determine the activation potential of the neuron.

   The threshold can also be represented by the bias. The bias is a further input, which transmits the constant value 1 and as weighting the value $b = -\theta$.

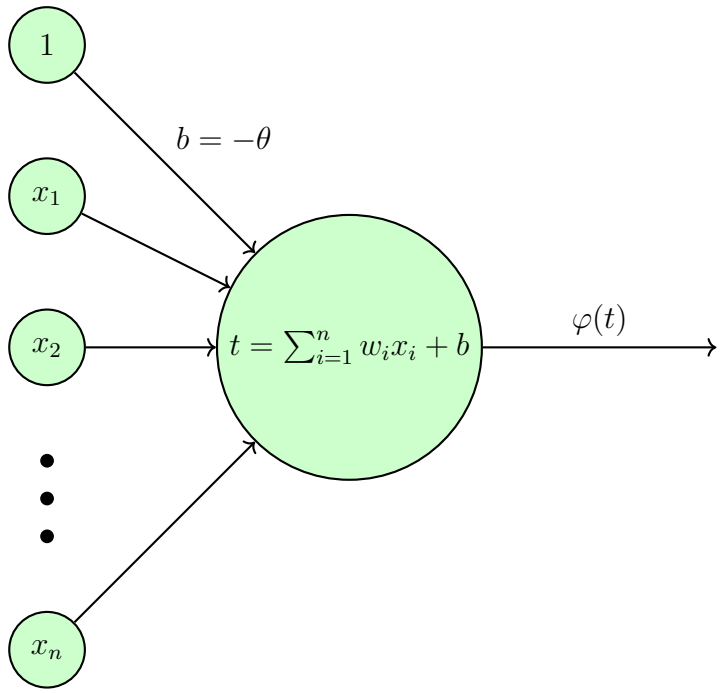

Figure 4: An artificial neuron with bias

4. The determined value of $t - \theta$ is used as an input in the activation function to determine the output of the neuron.

   From this process, the basic elements of a neuron can be determined:

   - Weighting
   - Threshold
   - Transfer function
   - Activation function

2.2.3 Construction of a neural network

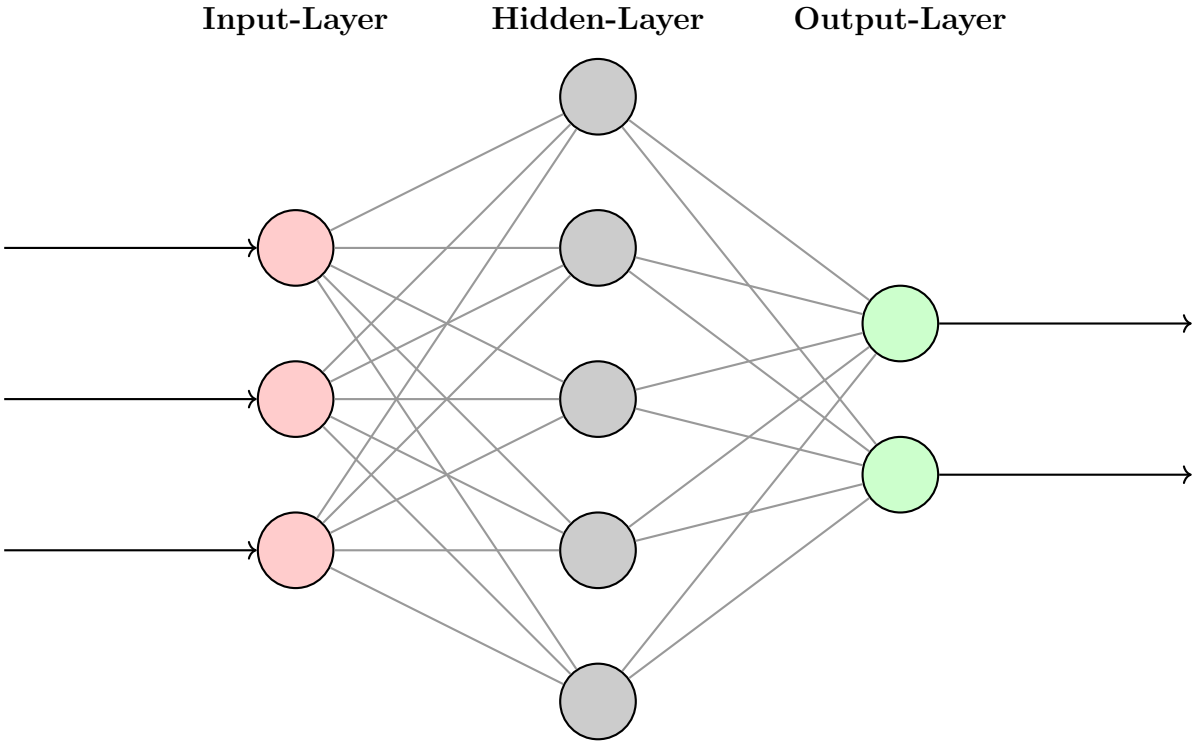

Figure 5: Neural network with hidden layer

Neural networks are built in several levels (layers). Each of these layers consists of a certain number of neurons and connections. Each neuron of a layer is connected to all neurons of the next layer via the already discussed weighted input connections.

- **Input layer:**

  The input layer is the first layer of a neural network. It contains the same Number of neurons, such as input and output values in the test and learning data available.

- **Hidden layer:**

  A neural network can not contain, one or more hidden layers. Neural networks can solve simple problems without hidden layers, many problems, such as the so-called "XOR problem", can't be solved without at least one of these layers.[36]

  The hidden layers consist of neurons, which together represent an internal representation of the input. The weighting to the individual neurons of the hidden layer is determined, for example, by means of back propagation.

- **Output-Layer:**

  The output layer is the last layer of a neural network. It delivers the results of the network to the outside world. The number of neurons in the output layer equals the number of expected output values.

2.2.4 Learning algorithms

Each neural network must be trained with training records. These records consist of rows with the input values and the expected output (supervised learning). The training itself is an iterative process in which each input and output passes through rows.[37]

In order for the network to learn how to approximate the output value, the weights and threshold of each neuron must be modified and adjusted.

The quality measure is the quadratic cost function:

$$E = \frac{1}{2} \sum_{i=1}^{n} (a_i - o_i)^2$$

The network error E consists of the accumulated squared deviations of the individual rows.

These are calculated from the difference between the expected output value a_i and the calculated network output o_i. The sum is multiplied by $\frac{1}{2}$. This factor simplifies the derivation. The aim now is to find the combination of thresholds and weights of the neurons in which the network error E is the lowest.

You could now let the net simply try all possible weights and thresholds until the combination with the smallest error E is found. This is not possible in practice, as there are usually too many parameters that have to be tried out within the network.[37]

For this purpose, learning algorithms are used instead. Learning algorithms distinguish between two categories, supervised and unsupervised learning.

2.2.5 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning, inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward.

The problem, due to its generality, is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics and genetic algorithms. In the operations research and control literature, reinforcement learning is called approximate dynamic programming, or neuro-dynamic programming.

The problems of interest in reinforcement learning have also been studied in the theory of optimal control, which is concerned mostly with the existence and characterization of optimal solutions, and algorithms for their exact computation, and less with learning or approximation, particularly in the absence of a mathematical model of the environment. In economics and game theory, reinforcement learning may be used to explain how equilibrium may arise under bounded rationality.[1]

While neural networks are responsible for recent breakthroughs in problems like computer vision, machine translation and time series prediction – they can also combine with reinforcement learning algorithms to create something astounding like AlphaGo.[13]

Reinforcement learning refers to goal-oriented algorithms, which learn how to attain a complex objective (goal) or maximize along a particular dimension over many steps; for example, maximize the points won in a game over many moves. They can start from a blank slate, and under the right conditions they achieve superhuman performance. Like a child incentivized by spankings and candy, these algorithms are penalized when they make the wrong decisions and rewarded when they make the right ones – this is reinforcement. Reinforcement algorithms that incorporate deep learning can beat world champions at the game of Go as well as human experts playing numerous Atari video games.

While that may sound trivial, it's a vast improvement over their previous accomplishments, and the state of the art is progressing rapidly.[13]

Reinforcement learning solves the difficult problem of correlating immediate actions with the delayed returns they produce. Like humans, reinforcement learning algorithms sometimes have to wait a while to see the fruit of their decisions. They operate in a delayed return environment, where it can be difficult to understand which action leads to which outcome over many time steps.

Reinforcement learning algorithms can be expected to perform better and better in more ambiguous, real-life environments while choosing from an arbitrary number of possible actions, rather than from the limited options of a video game. That is, with time we expect them to be valuable to achieve goals in the real world.[2]

2.2.6 Neural Networks and Deep Reinforcement Learning

Where do neural networks fit in with RL? Neural networks are the agent that learns to map state-action pairs to rewards. Like all neural networks, they use coefficients to approximate the function relating inputs to outputs, and their learning consists to finding the right coefficients, or weights, by iteratively adjusting those weights along gradients that promise less error.

In reinforcement learning, convolutional networks can be used to recognize an agent's state; e.g. the screen that Mario is on, or the terrain before a drone. That is, they perform their typical task of image recognition.

But convolutional networks derive different interpretations from images in reinforcement learning than in supervised learning. In supervised learning, the network applies a label to an image; that is, it matches names to pixels.

In reinforcement learning, given an image that represents a state, a convolutional net can rank the actions possible to perform in that state; for example, it might predict that running right will return 5 points, jumping 7, and running left none.

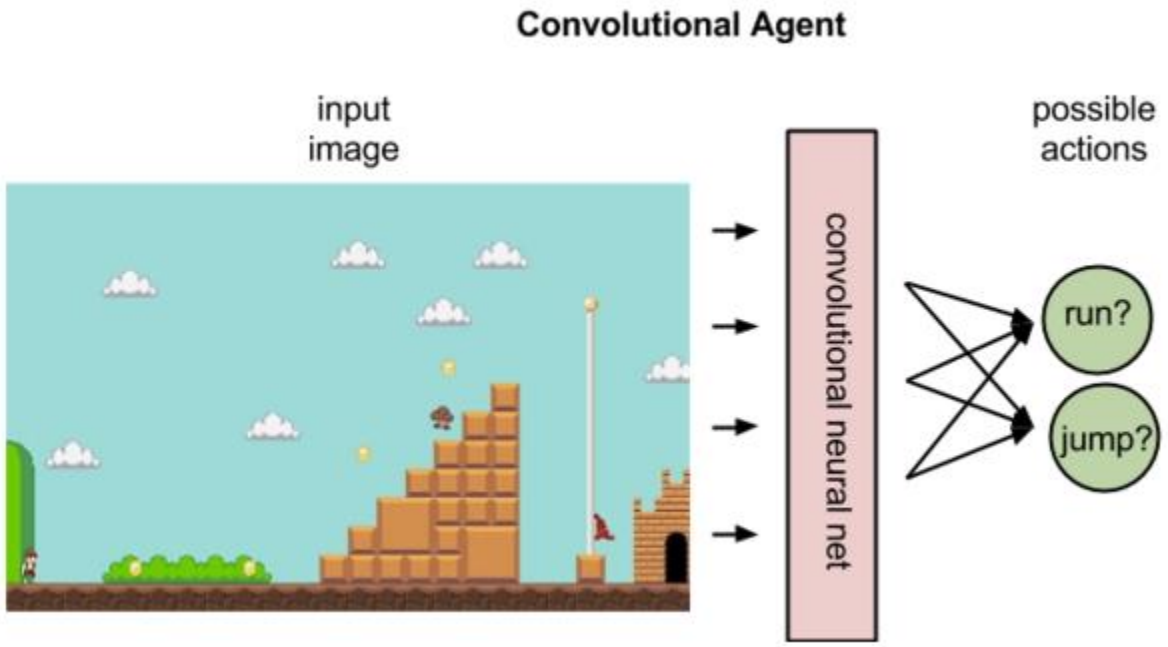

Figure 6: Convolutional Agent[2]

The above image illustrates what a policy agent does, mapping a state to the best action.

At the beginning of reinforcement learning, the neural network coefficients may be initialized stochastically, or randomly. Using feedback from the environment, the neural net can use the difference between its expected reward and the ground-truth reward to adjust its weights and improve its interpretation of state-action pairs.

Reinforcement learning relies on the environment to send it a scalar number in response to each new action. The rewards returned by the environment can be varied, delayed or affected by unknown variables, introducing noise to the feedback loop.[2]

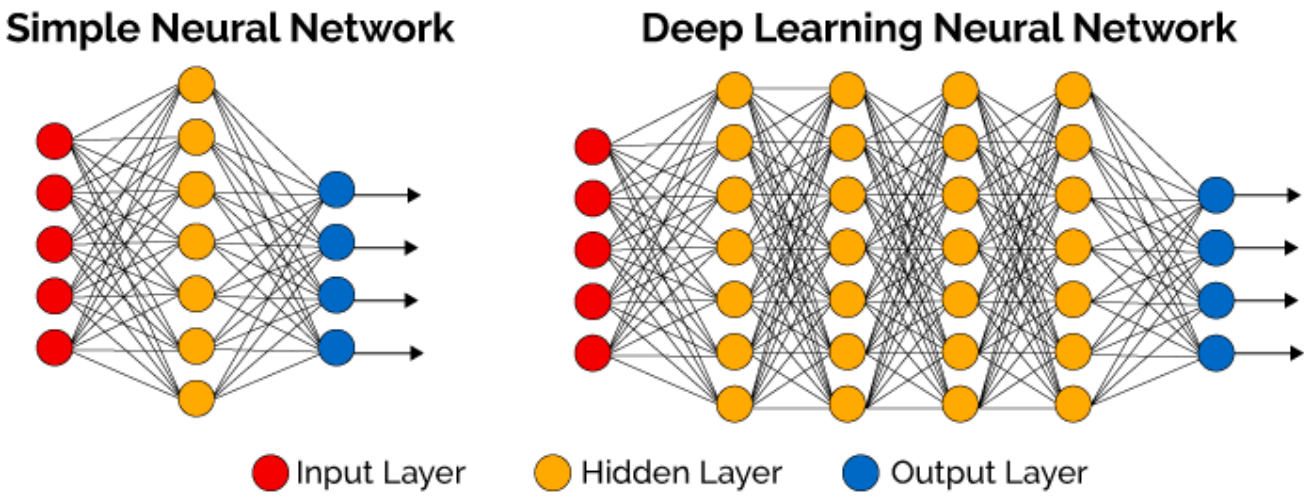

Figure 7: Difference between a Deep Learning Network and a Neural Network

The typical framing of a Reinforcement Learning (RL) scenario: an agent takes actions in an environment, which is interpreted into a reward and a representation of the state, which are fed back into the agent.

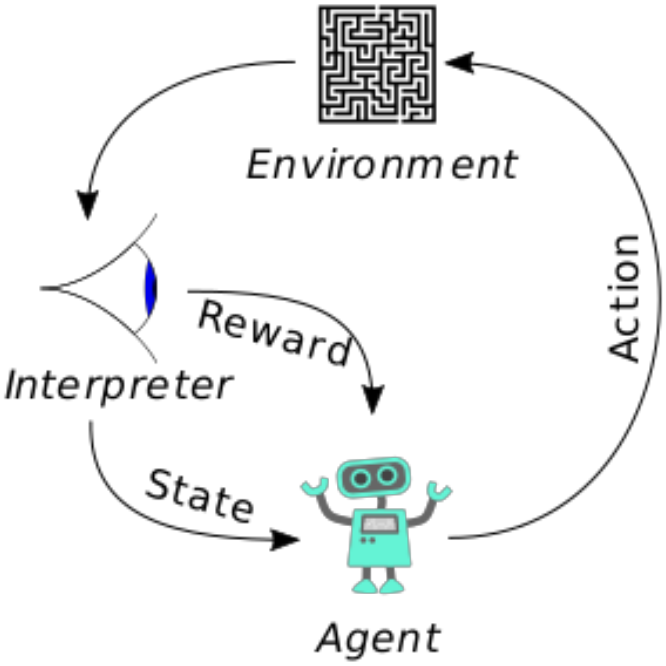

Figure 8: Reinforcement learning diagram[1]

2.3 OpenAI Gym

"OpenAI Gym is a toolkit for reinforcement learning research. It includes a growing collection of benchmark problems that expose a common interface, and a website where people can share their results and compare the performance of algorithms."[31]

OpenAI Gym is an easy to use and easy to integrate toolkit with a veriety of environments for reinforcement learning (RL) (cmp. 2.2.6) which we used for training our agent. An environment can be a retro arcade game (used in our project) or some other complex task which the agent should master.

```
# a small example program using OpenAI Gym

import gym
import random

# build gym environment. A full list of available environ-
# ments can be found at: https://gym.openai.com/envs/
environment = gym.make('LunarLander-v2')

# the range of actions an agent can perform. As a mathemat-
# ical set: [0..action_space[
action_space = environment.action_space.n

# mandatory first reset
environment.reset()

# generate random action (instead of predicted action from
# the agent)
action = random.randrange(0,action_space)

# observations provided by the gym environment after
# doing the random action in the environment.
observation, reward, done, info = environment.step(action)
```

Figure 9: A small example program using OpenAI Gym

Important for further understanding of how we utilized gym is the concept of episodes. An episode is a finite sequence of actions performed on the environment which either concludes in solving the environment (reaching a specific score which is different for each environment) or failing, basically representing a game played.

```
# example of an episode

# if the episode reaches this score the episode is finished
# successfully
score_solved = 200

# the score reached in this episode
score = 0

# perform the finite sequence of actions (terminated by
# the environment)
while True:

  # generate random action (instead of predicted action
  # from the agent)
  action = random.randrange(0,action_space)

  # observations provided by the gym environment after
  # doing the random action in the environment
  observation, reward, done, info = environment.step(action)

  # add the reward from the action to the overall reached
  # score of the episode
  score += reward

  # the done value provided by the environment is a Boolean
  # which specifies if the episode is finished (either suc-
  # cessfully or not)
  if done:
    if score == score_solved:
      print("finished episode successfully!")
    else:
      print("failed episode!")

    # reset environment after every episode (otherwise gym
    # throws an exception when calling env.step, since the
    # episode already terminated)
    env.reset()

    # return from episode loop
    break
```

Figure 10: Example of an episode (game) played on an OpenAI Gym environment

An agent has solved an environment if it succeeded x consecutive episodes. x is provided by Gym.

2.4 Tensorflow

2.5 Message Broker

A message broker is an intermediary computer program module that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver.

Message brokers are elements in telecommunication or computer networks where software applications communicate by exchanging formally-defined messages. Message brokers are a building block of message-oriented middleware (MOM) but are typically not a replacement for traditional middleware like MOM and remote procedure call (RPC). [21]

2.5.1 History

Message broker products are the middleware that embody different messaging solutions. In 1983 Vivek Ranadivé from Teknekron Software Systems began working on an idea based on the ideals of a Software Bus to enable applications to share data in a standard fashion. That work would later be referred to as "The Information Bus". Already in 1986 his ideas were put to use when Goldman Sachs, an American investment banking firm, launched cooperation with Teknekron to find solutions for the trading floor of the future.[22]

For nearly two decades the domain of message exchange was left for proprietary vendors and proprietary formats. Throughout the 1980s and 1990s message queuing kept evolving but in isolation since commercial message queue vendors strived for interoperability between client applications rather than worked on standardized interfaces for their message queuing products to utilize. [22]

The figure bellow shows a brief timeline of message queuing:

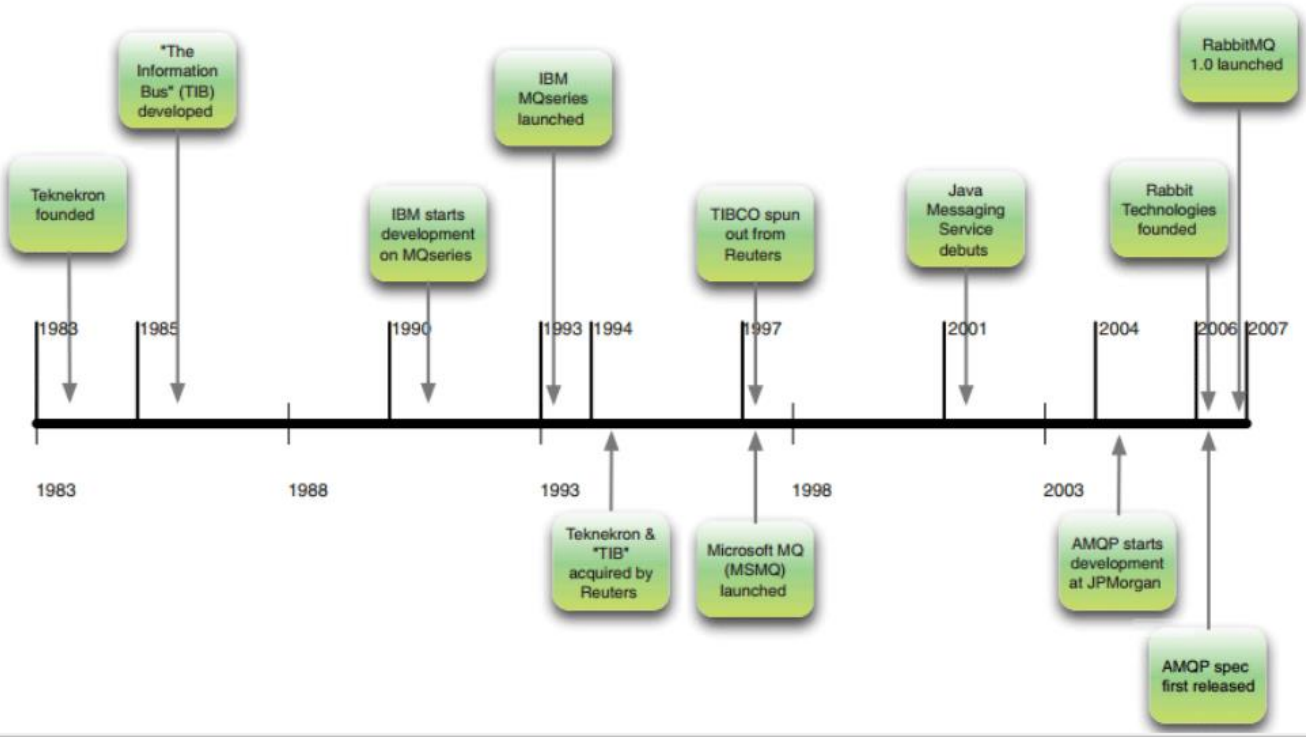

Figure 11: Timeline of message queuing

2.5.2 Functionality and architecture

A message broker is an architectural pattern for message validation, transformation, and routing.

It mediates communication among applications, minimizing the mutual awareness that applications should have of each other in order to be able to exchange messages, effectively implementing decoupling.[32]

The primary purpose of a broker is to take incoming messages from applications and perform some action on them. Message brokers can decouple endpoints, meet specific non- functional requirements, and facilitate reuse of intermediary functions. For example, a message broker may be used to manage a workload queue or message queue for multiple receivers, providing reliable storage, guaranteed message delivery and perhaps transaction management. [21]

Message brokers are generally based on one of two fundamental architectures: hub-and-spoke and message bus. In the prior, a central server acts as the mechanism that provides integration services, whereas with the latter, the message broker is a communication backbone or distributed service that acts on the bus.[23]

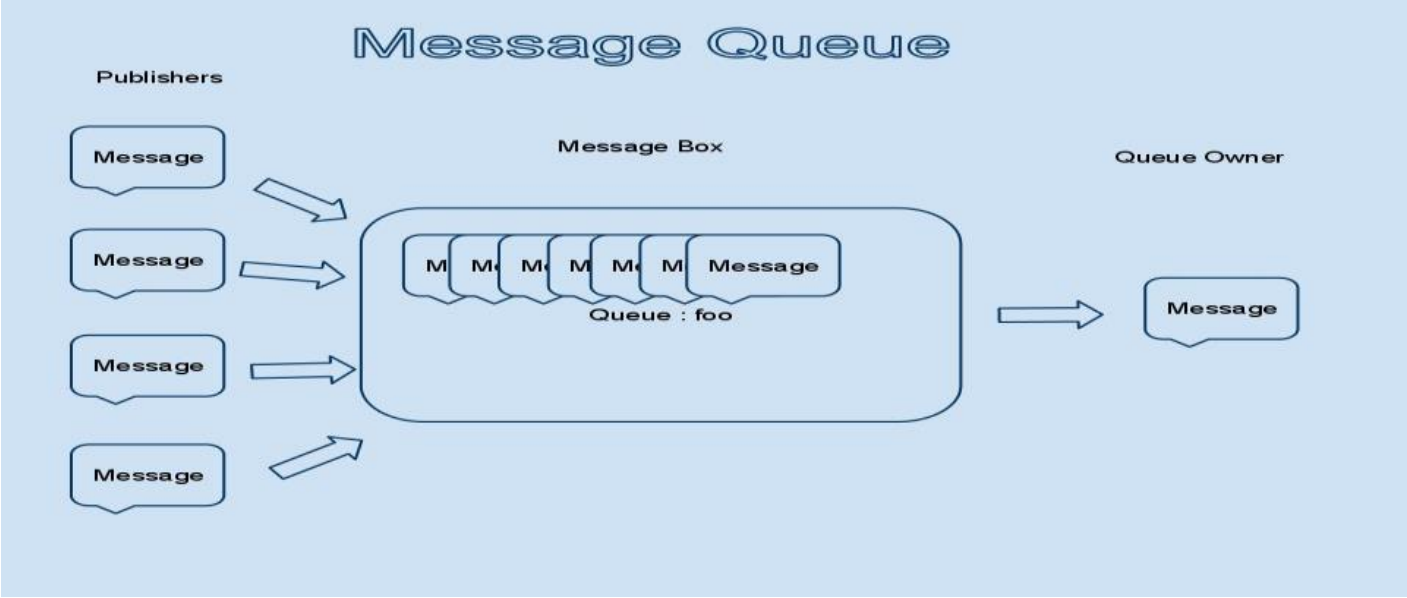

Figure 12: Scheme of a message queue

2.6 RabbitMQ

Figure 13: RabbitMQ Logo

For our project we used RabbitMQ as our messsage broker.

RabbitMQ is an open source message broker software (sometimes called message-oriented middleware) that originally implemented the Advanced Message Queuing Protocol(AMQP) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), Message Queuing Telemetry Transport (MQTT), and other protocols.[16]

The RabbitMQ server program is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and failover. Client libraries to interact with the broker are available for all major programming languages.[17]

2.6.1 Small glance on the history of RabbitMQ

Rabbit Technologies Ltd., originally developed RabbitMQ. Rabbit Technologies started as a joint venture between LShift and CohesiveFT in 2007[18], and was acquired in April 2010 by SpringSource, a division of VMware.[19] The project became part of Pivotal Software in May 2013.[20]

3 Application

Our goal was to make use of a lot of computing power in order to train our agent to master an OpenAI Gym Environment (cmp. 2.3). In this chapter we will document how we tried to achieve this goal with our distributed application.

3.1 The agent

Before going into detail on the application's architecture, here is a brief summary on what our agent actually is.

We programmed our agent as a neural network with the Keras library, which has an API for high level, high abstraction neural networks.

Keras uses Tensorflow as its backend for computations and basically only provides a nicer abstraction of Tensorflow.

```
# a small example program using Keras
#
# API documentation at: https://keras.io/

# Sequential is the keras object representing a neural net-
# work
from keras.models import Sequential
# a neural net is comprised of layers connected with each
# other. The Dense object represents a layer
from keras.layers import Dense

# the neural net. This specific neural net has four layers
# (the input (size of 12), two hidden (both 64 artificial
# neurons) and the output layer (size of 4)). The input
# layer does not have to be specified since the input_dim
# parameter of the first layer automatically generates the
# input layer.
model = Sequential([
  Dense(64, activation='relu', input_dim=12 ),
  Dense(64, activation='relu'               ),
  Dense(4,  activation='softmax'            ),
])

# define how the model should learn and some other meta
# information for the training process (learning algorithm,
# optimizer, etc)
model.compile(
        optimizer = 'adam',
        loss      = 'categorical_crossentropy',
        metrics   = ['accuracy']
)
```

```
# generate a dummy data set with corresponding labels with
# 1000 entries for training
import numpy as np

data   = np.random.random((1000,12))
# generating the labels (either a 0 or a 1)
labels = np.random.randint(2, size=(1000, 1))

# training the model, iterating 10 times
model.train(data, labels, epochs=10)

# using the neural net to predict the label of a random
# data point
test   = np.random.random((1,12))

model.predict(test)
```

Figure 14: A small example program using Keras

Our agent has the observation provided by the Gym environment (cmp. 2.3) as its input parameters and as output parameters the actions (the size of the output layer equals the action space a ($|[0..a[| = a$) of the Gym environment.

3.2 Architecture

This chapter will describe in detail how our application is build, how it works and what we use to achieve concurrency.

On a higher level our application is devided into two distinct parts, the Executor (E) and the Worker (W).

E is the part of our application that is doing the training and testing of our agent, while W generates training data which is sent via the RabbitMQ to E so E can train the agent with this data. After training E sends the agent to W so W can generate new training data with the updated agent.

This iteration is continued until the agent is able to solve the Gym environment (cmp. 2.3).

3.2.1 Network

Instances of both, E_i and W_j communicate over a message broker, in this case RabbitMQ (cmp. 2.5, 2.6).

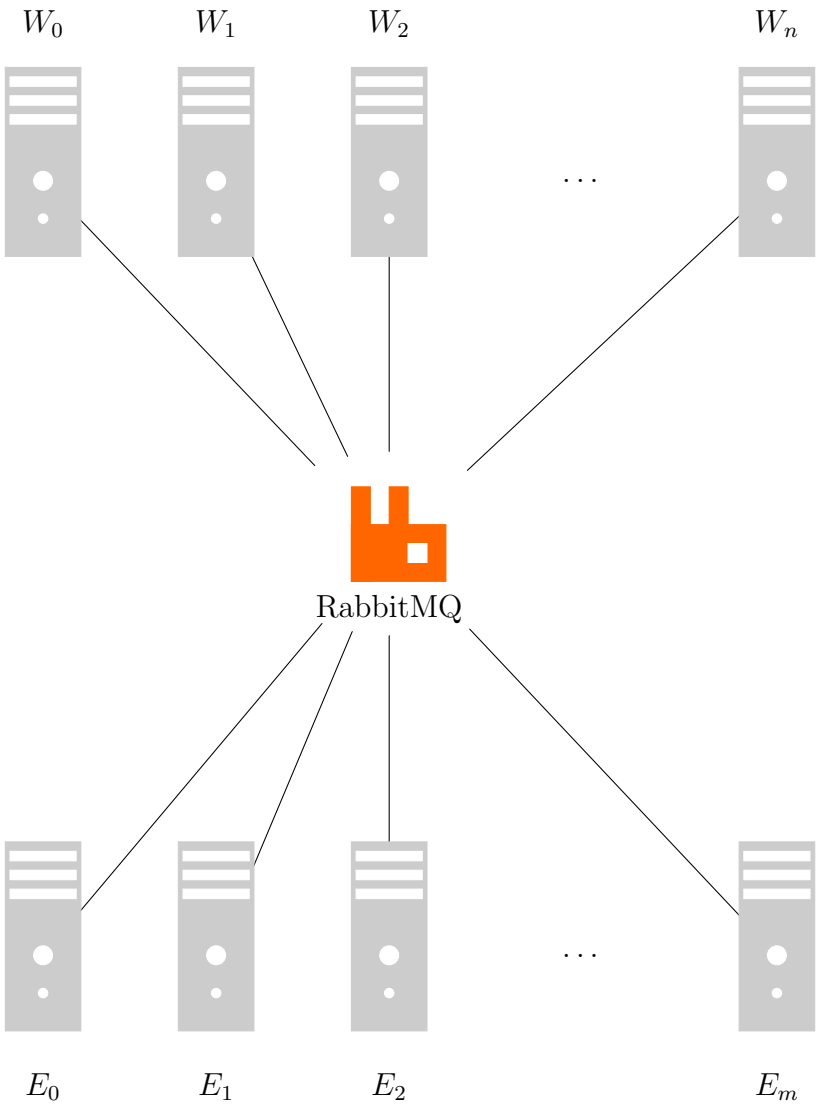

Figure 15: Architecture of our Application on the network level

3.2.2 Executor

The Executor E is the part of the application that is doing the training and testing of our agent. Training and testing is done incrementally in a loop, the "TTSL" (Training-Testing-Sending-Loop). An Executor instance E_i gets provided with its training data from the Worker instances W_j it is connected to (via the RabbitMQ).

If the testing part of the "TTSL" failed E_i sends the newly trained agent to every W_j (the sending part of the "TTSL"). Else if the testing was successful and the agent mastered the environment (cmp 2.3) it sends a message that testing was successful which kills every W_j.

The Executor is composed of three processes, the main process, the meta process and the TTSL process.

main process:

The main process first initializes global shared variables and constants it shares with the other two processes. Then it starts the meta process and the TTSL process.

After that the main process becomes a listener which listens for incoming data from the Worker instances connected to the RabbitMQ. If new data comes in it is saved in a shared variable so the TTSL process can access it.

TTSL process:

The process executing the TTSL.

meta process:

This process is a listener (like main becomes after initializing the shared memory and starting this and the TTSL process) which listens to a queue of the RabbitMQ so this Executor can communicate with the Workers it is connected to.

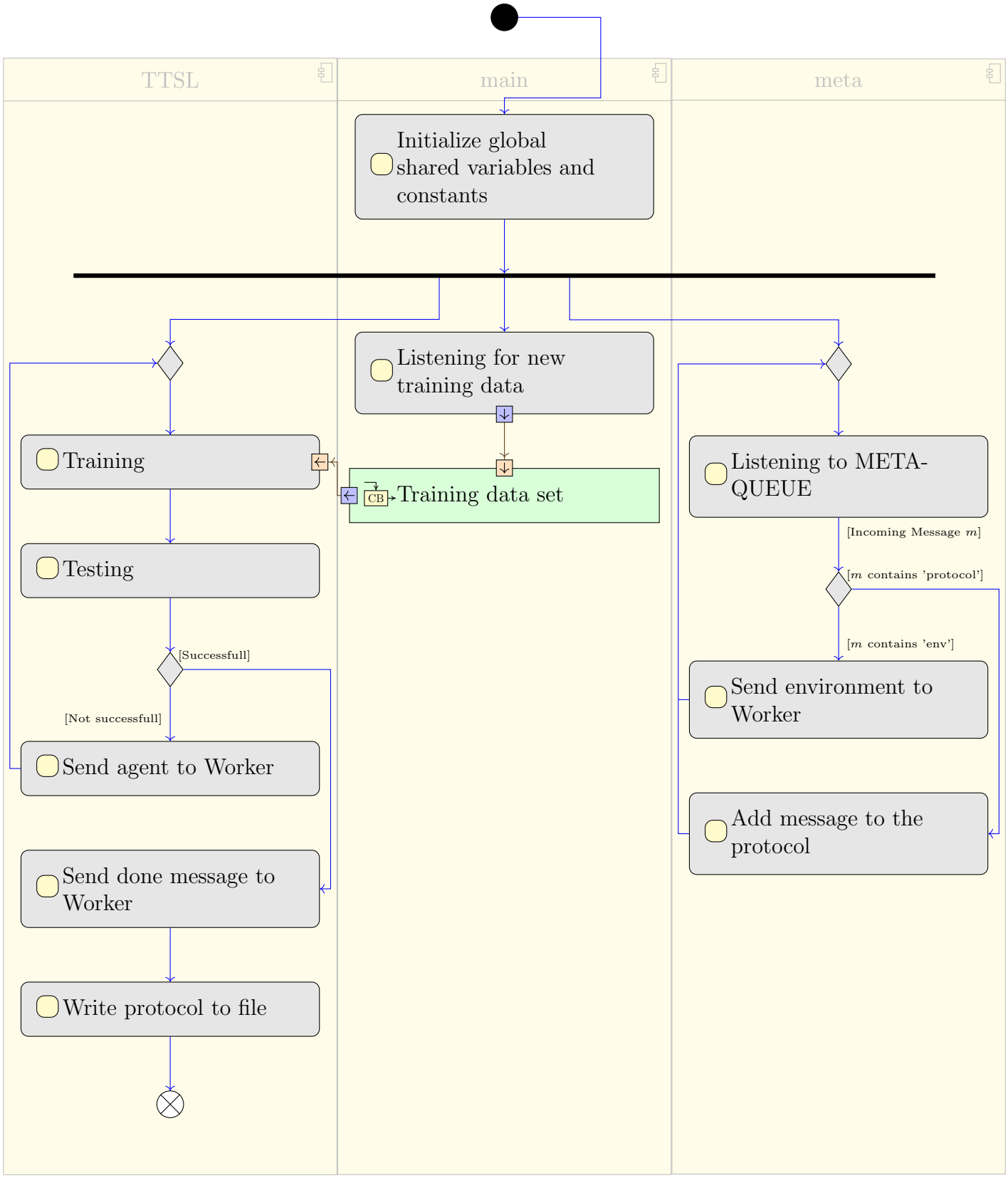

Figure 16: Activity diagram of the Executor

3.2.3 Worker

The Worker W is the part of the application that is generating the data set for training the agent. The generation of the training data set is the most expensive task when it comes to computational effort.

While doing the generation W is playing x many episodes consecutively. x is statically provided by us and is known at runtime. After finishing an episode the the score of this episode is decisive wether the episode is good enough for the training set, since the quality of the training data set is very crucial for the agent to succeed.

Quality management is done statically with some constants (like x).

Since generating training data is so expensive and can be done concurrently we use many processes controlled by a Python object called ProcessPoolExecutor from the concurrent.futures part of Python's standard library for this task.

```
# a small example program using a ProcessPoolExecutor
from concurrent import futures

# this is executed by every process. Every process gets
# an id (i) which is used as a factor for computing a power
# sequence in the range(100*i,100*i+100)
def power_sequence(i):
  start  = 100*i
  end    = start + 100
  primes = []

  for i in range(start, end):
    primes += i**i

  return primes

# the list of powers
powers = []

# the ProcessPoolExecutor that starts 10 processes which
# means a power sequence for range 0..999 is generated
with futures.ProcessPoolExecutor(max_workers=10) as e:
  # safe every process in fs
  fs = [e.submit(power_sequence,i) for i in range(10)]

  # await the return
  for f in futures.as_completed(fs):
    powers += f.result()

print(powers)
```

Figure 17: A small example program using ProcessPoolExecutor

Concurrent, multiprocessing and thread are the parts of Python's standard library wich provide rich features for concurrent programming.

While concurrent provides higher-level abstractions which are more easy to use, multithreading provides the more low-level, more powerful APIs for concurrent programming such as a standalone Process object, Locks (mutexes), Semaphores, Pipes, Queues or ProxyObjects (shared variables).

We use multiprocessing for spawning standalone processes, sharing variables and for mutexes (synchronization between processes).

Like the Executor (cmp. 3.2.2) a Worker instance is composed of more than one process. But while an Executor needs three, a Worker needs only two processes.

main process:

Like the Executor the Worker first initializes some static or shared variables and constants. But after the initialization part the worker has to get some information first before continuing execution.

Since the Worker should be able to run like a daemon waiting for his task (provided by an Executor) the Worker needs the information which environment he should generate test data for.

This information is provided by an Executor instance connected to the RabbitMQ. The Worker sends periodically a message to a queue the Executors meta process listens to. The Executor instance than answers with the name of the environment the Worker should use. The environment is specified by us when starting the Executor via its CLI.

While it is possible to connect many Executors to the RabbitMQ, they all can only be started with the same environment since otherwise corrupt data will destroy any chance of success (we did not implement a way to distinguish between different environments using the same RabbitMQ).

After having the environment the main process can continue.

The main process then starts the Worker's generating unit called "GSL" (Generate-Send-Loop). This is the loop corresponding to the Executors "TTSL" unit.

Followed by that the main process becomes a listener for a queue on the RabbitMQ. On that queue the main process gets the new agent provided by an Executor instance which is shared with the GSL process or a message which says that the agent succeeded. If that is the case the main process answers with a protocol to the queue the meta process of the Executor listens to and then kills the GSL process and itself.

GSL process:

The GSL generates and sanitizes the test data before sending it to the Executor. For that it uses Python's above mentioned ProcessPoolExecutor.

Since the Worker is not provided with an agent yet it generates the actions performed in every episode played randomly.

After the first batch of training data the Executor has processed, the Executor can send the first version of its agent to the Worker which can use it for generation afterwards.

It should be noted that the Worker not only uses the agent for generating new training data but also spawns some processes which generate training data randomly so the agent does not start to make the same mistakes over and over again.

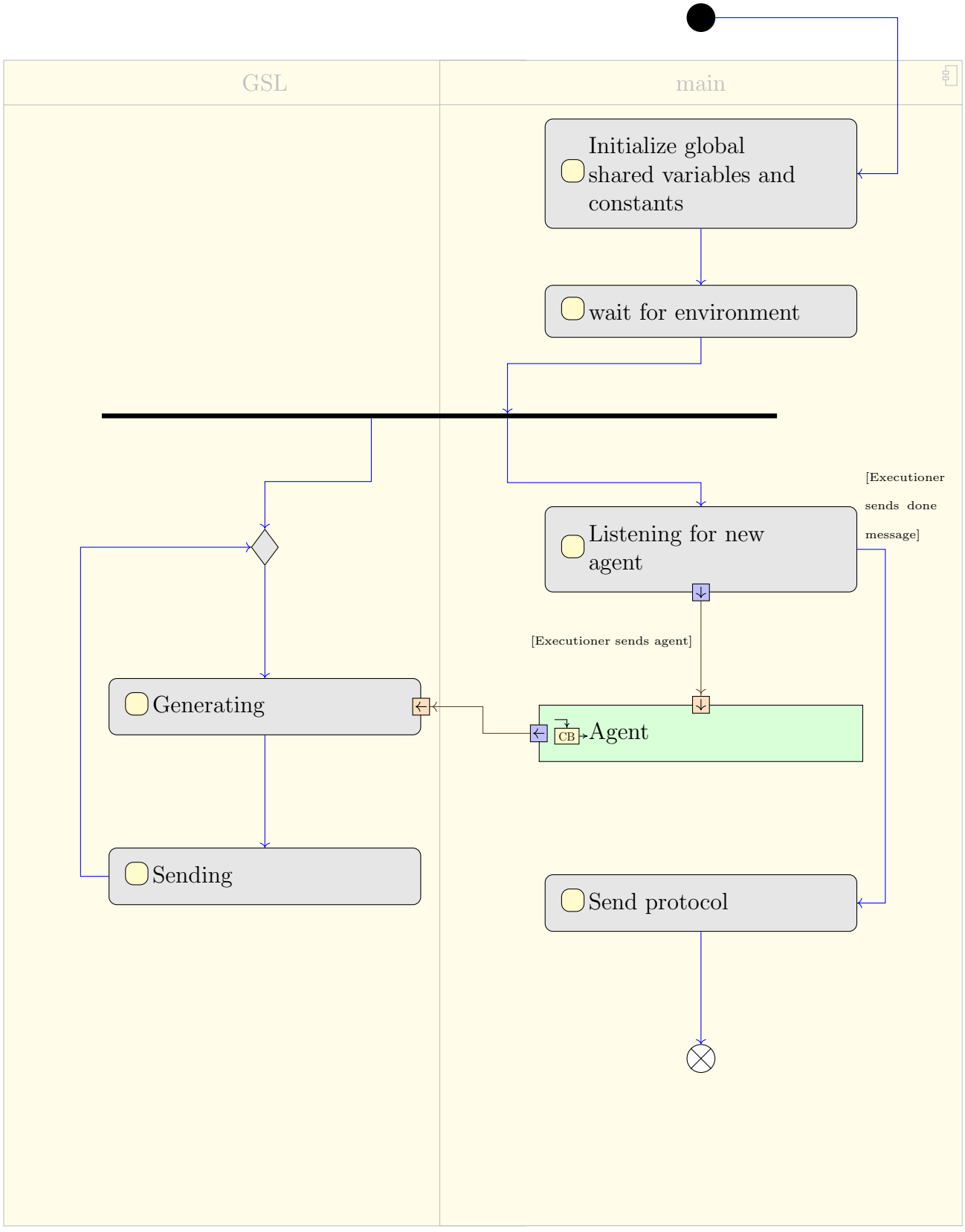

Figure 18: Activity diagram of the Worker

3.2.4 Queues and Exchanges

Now, this chapter will go more into detail on how we utilize the RabbitMQ.

First, there are two concepts we used for communication with the RabbitMQ called queues and exchanges.

queue:

This concept we already discussed in chapter 2.5.2. Now, for our project one property of a queue is interesting, which is "first come, first serve" or "FIFO" (First In, First Out), which means once a message is received by a listener (also called consumer), other listeners (consumers) that subscribed to this particular queue will never receive this message and all subscribed listeners are in a race condition for the next message.

Because of this we need another concept for distributing certain messages (e.g. sending our agent to each Worker, since every Worker should use a current version of the agent wich would not be possible if a Executor would send the agent to a queue, because then only one would receive the new agent instead of all Workers).

exchange:

For the above mentioned szenario we need a different message distribution called publish/subscribe.

Publish/subscribe message distribution can be achieved with an exchange provided by the RabbitMQ.

In this szenario the Executor would by the publisher while the Workers would be the subscribers. For every listener (subscriber) RabbitMQ generates a new queue and redistributes every message published to the exchange to the queues.

For this redistribution or routing are some methods available. We only used the method called fanout, which generates a copy of every published message for each listener (consumer) receiving on an exchange.

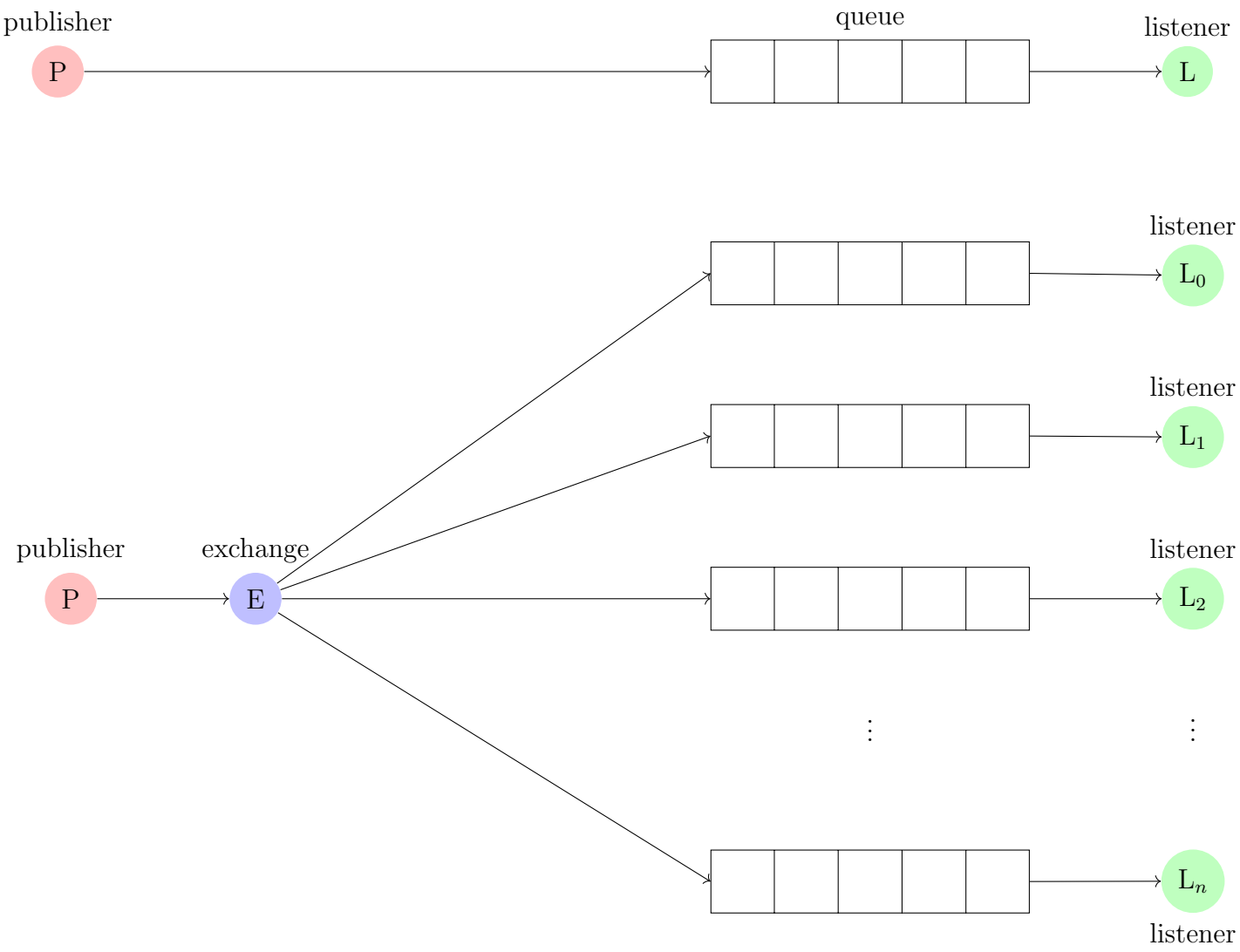

Figure 19: Scheme of a queue and an exchange

The queues and exchanges we used in our project:

meta_queue:

Queue the meta process of the Executor(s) is/are listening to. This queue is used by Workers to ask for the environment and for sending the protocol once the agent succeeded.

meta_exchange:

Corresponds to meta_queue. The Executor(s) is/are answering with the environment on this exchange.

data_queue:

Queue used by the Workers to send their generated training data to the Executor(s).

model_exchange:

Exchange the Executor(s) use(s) for sending the new agent to each Worker or, if the agent succeeded, for sending a message telling each Worker to send their protocols and shut down afterwards.

3.3 Results

First, we started developing our application for a Gym (cmp. 2.3) environment called "CardPole-v1".

In this environment the agent learns how to balance a stick moving a board on wich this stick stands either right or left.

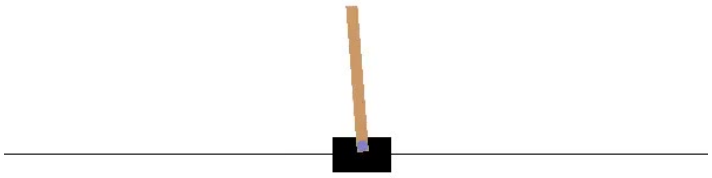

Figure 20: Screenshot of "CardPole-v1"

We developed our program on a HP EliteBook 8730w with a Intel Core 2 Duo processor (2.40 GHz) running OpenSuSE Leap 43. We needed an older Tensorflow version because the current version does not support this CPU anymore. Also Tensorflow was not able to utilize any GPU (Graphical Processing Unit), which means our developement environment does not have much computing power to offer.

The problem we faced with training an agent for "CardPole-v1" was that, running the program in our developement environment, we were able to build an agent that succeded this environment in at most three training loops (which overall took aproximately 5 minutes). This was achieved with running one Executor, the RabbitMQ and between one and three Worker instances on the same device which does not offer great computing abilitites (especially compared to the operational environment (the cluster in Room 1.242 containing 10 Apple Mac Pros)).

Since it would not made much sense running benchmarks on the operational environment training an agent to master "CardPole-v1" we needed a different Gym environment.

We decided we would take "LunarLander-v2", a Atari Arcade Game where the agent has to land an ufo safely inside a flagged area on the ground.

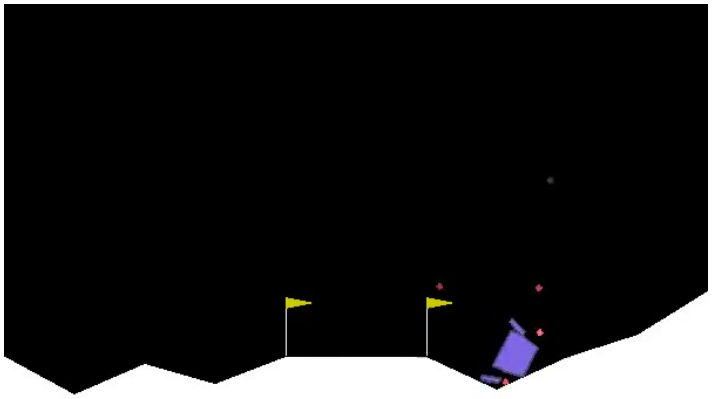

Figure 21: Screenshot of "LunarLander-v2"

"LunarLander-v2" is a far more complex environment to master since its action space is higher, it offers far more different observations and a more complex scoring system.

When we tried to train an agent that would master "LunarLander-v2" on our development environment we were never able to build an agent that would succeed in this (every time we tried running our program we canceled after four hours of runtime).

We did four test runs on our operational environment (in room 1.242), each with a different approach either in the amount of Executors (one or two) or the sanitation done to the training data (less strict which means more training data or more strict which means higher qualitiy of the training data).

For each test we used a neural network with two hidden layers, both with 64 artificial neurons.

1. This test run was done with a less strict policy toward our training data which means we took every episode that generated a score of over 0 and sanitized the episodes that had a score better than -400.

   Sanitation means we normalized every action's single score and took only the actions with a score better than 0.4 (we thought doing this we could cut out the part where the agent loses control over the ufo before crashing)

   We had 9 Workers running on 9 different machines and one Executor. Every Worker spawned 1 process generating training data with the agent and 5 processes generating training data randomly.

   Every process ran 1000 Episodes.

   This test run was terminated after approximately one hour.

   Observations:

   - 1000 Episodes were far too small. The Workers flooded the data_queue with messages containing only a small amound of data.
   - We discovered a very serious bottleneck. The messages containing the training data are formatted as a JSON string representation of a Python list which is parsed back into a list object when the Executor's main process receives it.

     But before training the list has to be parsed again into a format our agent (cmp. 3.1) understands. The list containing tuples with the observation and the corresponding action has to be splitted an parsed to a NumPy Array. This parsing is very expensive and while doing the parsing the Executioner's TTSL process holds the shared list object (which means

the main process has to wait until the TTSL process releases the list's mutex, which is one reason why the data_queue was so flooded).

2. This time we tried to conquer the issues we faced in the first test run. We used the same policy towards our training data but increased the amount of episodes played by each process. Every process doing generation randomly now did 10000 episodes while the ones using the agent played 5000.

   Also now we spawned 5 processes using the agent and 5 doing random actions.

   To avoid the bottleneck this time we added another Executor which roughly follows a reinforcement lerning technique called Double Q-Learing and is used to avoid overestimation for the training data (the agent doing the same mistakes over and over, because it trains with data it generated itself). [38]

   Both Executors are in a race condition for the next training data and both provide every Worker with its agent which we thought would lead to quality training data while keeping the data_queue small.

   We terminated this test run after approximately one and three quarters of an hour, after 11 iterations of the TTSL (both Executors). Both Executors combined had over 60 million data points and 40GB of more training data was still in the data_queue.

   Observations:

   - We only postponed the moment our application would kill itself with too much training data. Parsing again was too much.
   - Even though we had 60 million data points the agents were not able to show much progress. Not many test runs were able to exceed -150 (200 being the score an episode is played successfully).

3. After having again the problem of an overflowing data_queue and not much progress even after 11 iterations we tried a stricter policy. We took only episodes which generated a score of over 100 and sanitized the episodes which had a score over -50.

   Again we used to Executors and 8 Workers, all on different machines.

   We terminated this test run after approximately 45 minutes, both Executors having more than 16 million data points as their training data set.

   Observations:

   - At the first iterations we had some more success (scores over 100 which we were never able to reach before), but, at the end, even though the Executors both had 16 million data points, all on episodes with a score that had at least reached -50, both agents settled for test results around -150, like the two test runs before.

     We were not able to increase the performance of our agents.

4. We increased the strictness even more. We took only episodes that generated a score better than 150 and cut out the sanitation.

   This time we used only one Executor and again 9 Workers.

   Again we terminated after approximately 45 minutes, again unsuccessful.

   Observations:

- Before the workers can use the agent the agent has to be trained with 100 percent random generated data. Because the policy was so strict the first 6 received messages on the data_queue were empty. The seventh received message contained only 427 data points.
- Like the third test run this test run had even greater success at the beginning, once actually reaching 200. But again, even though only training with data better than 150 the agent settled at -150, only occasionally getting a better result around -50.

So, while we were able to create a distributed application that uses a modern approach to concurrency and was able to crunch a lot of numbers in a small amount of time, unfortunately we were not able to build an agent that could master "LunarLander-v2".

3.4 Where to go, what to do next

There are some points we would like to add to our application in order to make it able to build an agent that is good enough for "LunarLander-v2".

Statistical methods for data sanitation and a better agent:

Since we wanted to build an application that takes a task that takes a long time if done not concurrently and make it fast, we never really looked deeper into statistical methods from the fields of artificial intelligence research and data science that could have helped us build a better agent or a better training data set.

We thought we could generate enough data to outweigh the weaknesses our project shows when it comes to optimization of agent or training data set.

In the end our approach failed, so maybe by adding some optimizations of the agent or the training data set the application could be able to produce an agent that is able to succeed.

Some sort of load balancing:

We had hughe troubles in two test cases with an overflowing data_queue.

Our static approach (playing the same amount of episodes every iteration of the GSL (cmp. 3.2.3) did not work that well. At some time the Workers start to overwhelm the Executor(s).

To avoid this a unit doing load balancing should be added to the application that supervises the training data set generation using meta data from the Executor and the RabbitMQ.

Doing something with the parsing bottlneck of the Executor:

The parsing of the training data took longer than the actual training (optimized by Tensorflow).

Right now the parsing is done in a single process (the TTSL process (cmp. 3.2.2)), however we think the parsing could be optimized using a concurrent approach (e.g. worker processes like we use for the data generation (cmp. 3.2.3)).

Optimize the protocolling unit:

The problem with our current protocolling unit is, that it only saves the protocol when the program is finished successfully, which does not help when failing (like we did).

The protocolling unit should safe the data more often (e.g. could be integrated in the TTSL process (cmp. 3.2.2) or a new process only for protocolling could be spawned).

References

[1] Wikipedia. https://en.wikipedia.org/wiki/Reinforcement_learning#/media/File:Reinforcement_learning_diagram.svg. 17.07.2018.

[2] Artificial Intelligence Wiki. https://skymind.ai/wiki/deep-reinforcement-learning. 17.07.2018.

[3] The Making of Python. http://www.artima.com/intv/pythonP.html. 20.07.2018.

[4] A brief Timeline of Python. http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html. 20.07.2018.

[5] van Rossum, Guido. "[Python-Dev] SETL (was: Lukewarm about range literals). http://mail.python.org/pipermail/python-dev/2000-August/008881.html. 20.07.2018.

[6] Why was Python created in the first place? http://www.python.org/doc/faq/general/#why-was-python-created-in-the-first-place. 20.07.2018.

[7] Python 3.0 Release. http://python.org/download/releases/3.0/. 20.07.2018.

[8] PEP 3000 – Python 3000. http://www.python.org/dev/peps/pep-3000/. 20.07.2018.

[9] Python Definition. https://techterms.com/definition/python. 20.07.2018.

[10] what is python? http://www.pythonforbeginners.com/learn-python/what-is-python/. 20.07.2018.

[11] Benefits of Python. https://www.invensis.net/blog/it/benefits-of-python-over-other-programming-languages/. 20.07.2018.

[12] About python. https://www.python.org/about/. 20.07.2018.

[13] AlphaGo. https://deepmind.com/research/alphago. 17.07.2018.

[14] Python Programming language. https://en.wikipedia.org/wiki/Python_(programming_language). 20.07.2018.

[15] Itertools. https://docs.python.org/3/library/itertools.html. 20.07.2018.

[16] Which protocols does RabbitMQ support? https://www.rabbitmq.com/protocols.html. 20.07.2018.

[17] RabbitMQ. https://en.wikipedia.org/wiki/RabbitMQ. 20.07.2018.

[18] Launch of RabbitMQ Open Source Enterprise Messaging. http://www.rabbitmq.com/resources/RabbitMQ_PressRelease_080207.pdf. 20.07.2018.

[19] Rabbit Technologies announce acquisition by SpringSource. https://web.archive.org/web/20100418132113/http://www.rabbitmq.com/news.html. 20.07.2018.

[20] Proudly part of Pivotal. https://web.archive.org/web/20130602054940/http://www.rabbitmq.com/news.html. 20.07.2018.

[21] Message Broker. https://en.wikipedia.org/wiki/Message_broker. 20.07.2018.

[22] Message brokers and RabbitMQ in Action. https://www.theseus.fi/bitstream/handle/10024/78263/Kamppuri_Tsuri.pdf?sequence=1. 20.07.2018.

[23] Integration Brokers and Web Services. https://books.google.de/books?id=2EonCgAAQBAJ&pg=PA71&redir_esc=y#v=onepage&q&f=false. 20.07.2018.

[24] General Python FAQ. http://docs.python.org/faq/general.html#what-is-python-good-for. 20.07.2018.

[25] What is Python? Executive Summary. http://www.python.org/doc/essays/blurb/. 20.07.2018.

[26] General Python FAQ. http://www.python.org/doc/faq/general/#what-is-python. 20.07.2018.

[27] Contracts for Python. http://www.nongnu.org/pydbc. 20.07.2018.

[28] Contracts for Python. http://www.wayforward.net/pycontract/. 20.07.2018.

[29] The Making of Python. http://www.artima.com/intv/pythonP.html. 20.07.2018.

[30] PyInstaller Home Page. https://www.pyinstaller.org/. 20.07.2018.

[31] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI Gym, 2016.

[32] Ejsmont. "Asynchronous Processing". Web Scalability for Startup Engineers. McGraw Hill Professional, 2015.

[33] Erdmann, A., Erdmann, U., and Martens, A. Grüne Reihe. Materialien für den Sekundarbereich II. Schroedel Verlag GmbH, Braunschweig, 2004.

[34] Hoettinger, and Raymond. PEP 289 - Generator Expressions. Python Software Foundation, 2002.

[35] Nosrati, M. Python: An appropriate language for real world programming.

[36] Rey, G. D., and Beck, F. Neuronale Netze - Eine Einführung. http://www.neuronalesnetz.de/downloads/neuronalesnetz_de.pdf. 17.07.2018.

[37] Trinkwalder, A. Netzgespinste. c't 6 (2016), 130–135.

[38] van Hasselt, H., Guez, A., and Silver, D. Deep Reinforcement Learning with Double Q-learning. https://arxiv.org/pdf/1509.06461.pdf. 23.07.2018.