Learning representations versus feature crafting in deep reinforcement learning

Anssi Kanervisto

Master's Thesis

UNIVERSITY OF
EASTERN FINLAND

School of Computing

Computer Science

May 2017

ITÄ-SUOMEN YLIOPISTO, Luonnontieteiden ja metsätieteiden tiedekunta, Joensuu
Tietojenkäsittelytieteen laitos
Tietojenkäsittelytiede

Opiskelija, Anssi Kanervisto: Esitysten oppiminen versus piirteiden kehitys syvässä vahvistusoppimisessa
Pro gradu -tutkielma, 73 s., 2 liitettä (2 s.)
Pro gradu -tutkielman ohjaajat: FT Ville Hautamäki ja FT Wojciech Jaśkowski
Toukokuu 2017

Tiivistelmä: Kone-, syvä- ja vahvistusoppimista käyttävät tietokoneohjelmat ovat mahdollistaneet monimutkaisemmaskin ympäristöissä toimivien agenttien luomisen. Erityisesti peleille luodut agentit ovat voittaneet parhaimpia ihmispelaajia. Näihin peleihin kuuluu klassisia mutta vaikeita pelejä kuten shakki, Go ja Pokeri. Yksinkertaisemmatkin videopelit voivat tarjota haastetta näille agenteille johtuen niiden visuaalisen monimuotoisuudesta ja osittain havaittavasti tiloista, esim. shakissa pelaaja näkee koko laudan, video pelissä ei välttämättä. Esimerkiksi ihmispelaajaa parempi agentti Atari pelejä varten kehitettiin vasta n. 30 vuotta varsinaisten pelien julkaisun jälkeen.Syvän vahvistusoppimisen agentit voivat käyttää raakaa harmaasävykuvaa pelistä, samaa mitä ihmispelaaja näkee, toimiakseen ympäristössä. Tämä ei vaadi erillistä piirteiden irroittamista ja on täten helppo tapa antaa syötettä agentille. Tämän tutkielman tarkoitus on tutkia onko kyseinen harmaasävykuva myös tehokkain syöte, eli antaako se agentille mahdollisuuden oppia parempia toimintatapoja kuin mm. erikseen irrotetut piirteet. Harmaasävykuva yksinään voi tarjota tietoa joka ei ole relevanttia ongelman ratkaisun kannalta (mm. pienet yksityiskohdat seinässä), mutta kehittyneempi piirteiden irrotus voi hävittää tietoa joka olisi hyödyllistä agentin oppimisen kannalta. Tämä on osittain seurausta siitä miten piirteet usein ovat datamäärältään pienempiä kuin raaka data, mutta on mahdollista että agentti voisi oppia raa'asta syötteestä jotain mitä kehittäjät eivät ole huomanneet.

Avainsanat: vahvistusoppiminen, syväoppiminen, piirteiden irroitus, esitystavan oppiminen, koneoppiminen, neuroverkot, videopeli

ACM-luokat (ACM Computing Classification System, 2012 version): A.m, K.3.2

- **Computing methodologies ~Reinforcement learning**
- **Computing methodologies ~Feature selection**
- *Computing methodologies ~Neural networks*
- Computing methodologies ~Artificial intelligence

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry, Joensuu
School of Computing
Computer Science

Student, Anssi Kanervisto: Learning representations versus feature crafting in deep reinforcement learning
Master's Thesis, 73 p., 2 appendices (2 p.)
Supervisors of the Master's Thesis: PhD Ville Hautamäki and PhD Wojciech Jaśkowski
May 2017

Abstract: Machine, deep and reinforcement learning have made it possible to create artificial agents that can function in a complex environments to complete a given task. Especially agents created for different games have been able to win the best human players. This includes classical but computationally difficult games such as chess, go and poker, but recently there has been similar achievements with video games. Simple video games also offer challenge for artificial agents due to their relatively complex visuals and partially observable states, e.g. in chess a player can see the whole board, but in first-person shooter player only sees virtual avatar's point-of-view. For example, a super-human agent for a set of Atari was created circa 30 years after release of the games. Deep reinforcement learning agents are able to use same visual input human player would perceive, which can be gray-scale or color image of the screen. This avoids the issue of feature extraction in favor of learning representations, and is an easy way to provide input features for the agent. The purpose of this thesis is to study if the raw input is also the highest performing way of providing input features, compared to using feature extraction. Gray-scale image can provide information that is not directly apparent while designing feature extraction method, and thus is lost during feature extraction as features require less space. It is possible that agent's neural network is able to learn something unforeseen and useful from the raw input, which increases the performance of the agent.

Keywords: reinforcement learning, deep learning, feature extraction, learning representations, machine learning, neural networks, video game

CR Categories (ACM Computing Classification System, 2012 version): A.m, K.3.2

- **Computing methodologies ~Reinforcement learning**
- **Computing methodologies ~Feature selection**
- *Computing methodologies ~Neural networks*
- Computing methodologies ~Artificial intelligence

List of abbreviations

w.r.t	"with respect to"
SGD	Stochastic gradient descent
MDP	Markov decision processes
POMDP	Partially observable Markov decision processes
DQN	Deep Q network
CNN	Convolutional neural network
RNN	Recurrent neural network
LSTM	Long-short term memory
TD	Temporal difference
CPU	Central processing unit
GPU	Graphical processing unit
A3C	Asynchronous advantage actor-critic
GA3C	GPU asynchronous advantage actor-critic

List of common symbols

Scalar-valued variables are written in small letters (e.g. x). Vectors are written in small letters and bold-faced ($\boldsymbol{x}$), and matrices are in capital letters and bold faced ($\boldsymbol{A}$). Unless otherwise mentioned, vectors are **column-vectors** and matrices have **row-vectors**, i.e. rows of matrix are observations and columns are feature components. Subscript notion either means indexing of vector/matrix or different versions of same variable, e.g. at different time steps.

$t \in \mathbb{N}$	Time step.
$T \in \mathbb{N}$	Final time step, time step of the terminal state.
S	Set of all possible states.
A	Set of all possible actions.
$s \in S$	State provided by an environment.
$a \in A$	An action to be executed in an environment.
$P(s_{t+1} \mid s_t, a_t)$	Model of state transitions in the environment.
$R(s_t, a_t, s_{t+1})$	Reward function, returns rewards r_t for given state transition.
$r \in \mathbb{R}$	Reward from executing action in an environment.
e	An experience. Equals to (s_t, a_t, r_t, s_{t+1}) at given t.
$\pi(s_t)$	Policy function, returns an action a_t given a state.
$\pi^*(s_t)$	Optimal policy function.
$V_\pi(s_t)$	Value function under given policy.
$Q_\pi(s_t, a_t)$	State-action value function under given policy.
$\gamma \in \mathbb{R}$	Discount factor, controls how far-sighted reinforcement learning is.
$N \in \mathbb{N}$	Number of observations / feature vectors.
$d \in \mathbb{N}$	Number of components in feature vectors (dimension).
θ , w	Model parameters/weights.
x	Input feature(s).
y	True output(s) (also called "ground truth", "target").
$\hat{y}$	Estimated output.
$\eta \in \mathbb{R}$	Learning rate (often a small constant, e.g. $[10^{-8}, 10^{-1}]$).

Contents

1 Introduction

Creating agents that interact with an environment to solve a problem have been studied for years by using different methods, including algorithms (e.g. tree search), genetic algorithms [13] and reinforcement learning (e.g. [41]) . The appearance of so called *deep learning* (e.g. [15]) has had an impact on creating intelligent agents, one of the examples being recently proposed *Deep Q Network (DQN)* [27] which was able to beat human players in number of Atari games by only using the same inputs human would receive.

The key aspect of deep learning is to avoid manual feature crafting by letting computer *learn representations* from the raw, possibly noisy and flawed input data. Images can be fed into system directly instead of first extracting expert features for the image, for example. This is proven to be effective method in many tasks [15], for example in detecting and captioning objects from an image [50], recognizing what is being spoken [16] and chatting with a human in a natural manner [47].

If the task is to build a system that acts in an environment optimally by some metric, *reinforcement learning* [41] provides a framework build systems for deciding actions given inputs. These methods are mostly illustrated and tested with toy examples where the environment can be represented with small set of discrete values in tabular format. For example [41] uses simple boards and theoretical games like n-armed bandit problem, and [14] describes relatively simple pole-balancing problem. Combining reinforcement learning with methods from area of deep learning, *Deep Q Network* (DQN) [27] was able to use raw image data from an Atari game without manual feature extraction and beat human player in most of the tested Atari games. Since publication of DQN a number of improvements to original DQN have been proposed (e.g. [48, 32, 46]) including a number of different reinforcement learning methods for similar video game based environments.

Number of platforms for creating and/or evaluating reinforcement learning agents in richer environments has been also published, like OpenAI Gym [7], ViZDOom [22], DeepMind Lab [3], Minecraft [20] and OpenAI Universe [31]. Mentioned platforms **use video games designed for human players as a platform for AI agents.**. One could argue this signs of an improvement in such decision-making agents as we can start to apply them to increasingly challenging tasks, especially to ones that are origi-

nally created for humans.

However, while reinforcement learning systems using deep learning with raw sensory inputs with success, like DQN with Atari games [27], there is little research on the subject if raw information such as images is beneficial over higher level features (e.g. [5]). Arguably both approaches have their cons and pros: Using deep learning and raw image is straight forward "plug-and-play" and can yield good performance, as long as the model itself is large enough and one gives enough time and data to train the system. However, using manually defined features can incorporate researcher's domain knowledge to the system and the crafted feature vectors can be smaller, allowing faster training. Then again, process of extracting features can discard some essential information that could be beneficial to the machine learning system, while deep learning receives most if not all of the information available.

This thesis aims to find some answers to whether raw data and deep learning is a better approach than manually crafted features. Specifically, the research question is **"Do reinforcement learning agents benefit from using raw image data compared to using other features?"** Hypothesis is that using raw information allows reinforcement learning agent to perform better than using higher level features. This thesis studies this with experiments with reinforcement learning agents in a 3D environment ViZDoom [22]. This thesis first introduces general machine learning and reinforcement learning basics, including terms and methods. Next is introduction to deep learning and how it can be combined with reinforcement learning (more precisely, neural networks were combined with reinforcement learning). After theoretical background thesis describes experimental setup to approach the research question and reports the results along with a discussion of the results.

1.1 Video games for artificial intelligence research

Developing decision systems, such as the reinforcement learning agents, requires an environment and a task to work on. This can be provided by simulations for a specific task (e.g. locomotion), board games (e.g. Go, Chess) or lower level toy tasks (e.g. pole-balancing task described in [14]). By implementing the environment for a specific task researchers are able to obtain necessary values from the software interface which is then fed to the decision agent. However, with the development of deep learning we can

also use raw sensory input such as RGB or gray-scale images. This allows extending set of possible environments to video games which do not provide the direct values, but instead provide an image and audio. This is utilized in the experiments of this thesis by using video game "Doom" [18] as the environment with different tasks.

Some video games have been built to emulate physical world in terms of visual fidelity (e.g. lighting, shadows, colors) and interaction (e.g. modeling physics). For this reason they provide rich and complex set of environments which were originally designed for human players. They are also convenient to use compared to real life measurements and robotics because they offer a mostly noiseless, controllable environment. This is especially useful with deep reinforcement learning (see Chapter 4) which can require hundreds of hours of interaction with the environment to optimize parameters of a neural network. This can be sped up by running the video game at a faster rate or by running multiple instances of the game in parallel, which is less feasible for the physical world. While current video games do not provide as complex environment as our physical world, modern games published by big companies are approaching photo-realistic levels in terms of visual fidelity. For example, popular video game "Grand Theft Auto V" [30] has been utilized to produce a dataset of images for training neural networks [35].

Video games and simulations also provide platforms for creating different tasks very similar to what we could create in physical world. Previously mentioned "Grand Theft Auto V" could also be used to create a setup for a self-driving car. Since the game includes simulation of traffic and detailed roads, researchers could try out their methods in this simulated environment first to get a rough idea how their methods could work in a physical world scenario. If the video game is rich in features, one could experiment with general artificial intelligence that could solve multiple different problems, such as completing game's missions without pre-defined help. Outside academic publications, there are videos of "Grand Theft Auto V" being used for training a self-driving car [34] (links to the DeepDrive videos: `https://www.youtube.com/watch?v=1uURlRKfLqY` and `https://www.youtube.com/watch?v=X4u2DCOLoIg`).

Video games also provide a connection between humans and artificial agents. The previously mentioned "Grand Theft Auto V" provide multiplayer feature that allows multiple human players to play in same world. We could replace some of these players with artificial agents and directly interact with the agents this way. Sharing similar plat-

form also supports measuring performance of an artificial intelligence against human players, like in case of Deep Q Network [27] where researchers were able to gather performance scores of professional human players and the built artificial system. Some games, like "Starcraft 2" [11] and "Dota 2" [8], are used in professional, competitive tournaments akin to sports ("e-sports"). Similarly to sports, e-sports have a number of professional players who can provide good basis for measuring *super-human performance*, as they are proven to be most capable human players in the game via official tournaments.

While competing against human players in modern (between 2010 and 2017) is interesting, executing said games requires a lot of computing power and technical setup to be achievable. Instead, video game "Doom" [18] provides a first-person shooter style of setup with first-person view and a 3D environment (see Figure 1). VizDoom [22] is built on-top of Doom to provide means to train and evaluate artificial agents by running the game at high thousands of frames per second or even in parallel. This thesis will use VizDoom for empericial experiments.

Figure 1: *An image of Doom [18] video game (via ViZDoom [22]). The generic goal is to kill enemies and stay alive using available pickups and weapons. Player can move around in 3D space and turn left and right (ViZDoom also allows looking up and down). While game is visually simple it still provides challenging task for bots to use as a visual input as of writing this thesis. Image source:* `http://vizdoom.cs.put.edu.pl/`

2 Reinforcement learning

Reinforcement learning [41] is about learning to make actions in an environment which maximizes a numerical reward. It encompasses methods, definitions and framework to work on natural decision problems. "Natural", because this is what we humans and other animate entities do all the time in our physical world. We constantly have to make decisions, larger and smaller, which somehow benefit us in a longer or shorter term. Humans eat when they are hungry to satisfy the hunger, spend time at a job to gain wealth to ease live in future and study for new subjects to possibly get a paying job later in future [citation needed].

We can recreate some of this artificially. For rough examples: A hypothetical vacuum-cleaner robot has to decide in which order to clean the rooms/areas to minimize distance traveled. An artificial video game player (bot) decides where to go next to find possible enemies or do other objectives defined by the game. A self-driving car must constantly make small but crucial decisions while driving based on its surroundings to avoid hitting pedestrians or other vehicles in the traffic.

Some of the tasks for bots can have simple solutions. For example, since we know all the rules of the chess, and the gaming board is strict grid of known pawns, we can program computer to go through all possible states and moves made by them and their enemy. By going through all possible states till game ends, algorithm can compute which action in current state is the most likely to lead to a victory (or which action is least likely to result to defeat). In theory this works well, but the number of computations required to go through all states might be simply too high for any computer solve in a meaningful time.

Another approaches exist for such decision problems. For example genetic algorithms [13] use fitness function to evaluate slightly mutated versions of some system and select the highest performing systems for the next generation [13]. One can also remove learning part completely and use methods like tree-search, where algorithms searches tree made of states (nodes) and actions (edges) to find a way to reach a good state. One recent example of this was AlphaGO [39] which used Monte-Carlo tree-search to find good actions, and managed to beat world-master in game of Go.

For the purpose of this thesis we focus on reinforcement learning. While there is no

clear, best-performing approach currently, recent advances like Deep Q Network [27], AlphaGo [39], Asynchronous reinforcement learning [26] provide empirically good basis to work on, especially in video games in a end-to-end setting (raw image in, action out).

2.1 Core concepts

We require few concepts before we can properly define the problem. First, we call our decision making system an *agent* [41]. Agent encompasses the means to make decisions of actions it wants to take in the environment. Second, we need a space which can include our agent and define our task. We call this an *environment* [41]. According to [41] environment is everything that is outside of an agent. I.e. even if environment in reality has other "agents" (e.g. opponent player), they are included in the environment our agent is interacting with. Finally we can include our *task*. The task defines what is agent's mission in the provided environment, and agents job is the start making decisions and executing actions to complete the provided task. More formal definitions will be presented in Chapter 2.2

Instead of using predefined dataset with known or unknown targets like in machine learning (see Chapter 3), reinforcement learning approaches task by learning from interaction with the environment [41]. Supervised learning uses given set of data to optimize the parameters with respect to some loss/target function, while reinforcement learning has to gather this data via interaction and learn itself what is desired output given some input. Instead of known targets/outputs the reinforcement learning uses *reward* [41] to optimize the parameters of the agent to generate optimal outputs in the future. Reward is a scalar value given by the environment after executing an action from the agent which tells how good the executed action was. It should be noted that the reward function might not explicitly model the final goal of the task. For example, we can give reward for sub-objectives that benefit reaching to final goal.

Finally, the task for an agent is quite simple: **Interact with the environment so that sum of obtained reward is maximized**. The formulation of this task the agent also accounts for the possible future situations and rewards instead of just considering the very next actions, which would be very short-sighted. In our example of game of chess, we could give agent positive reward for each pawn they eat from opponent

(or from winning the game). The agent would learn to eat opponent's pawns since it grants immediate reward, but it would also learn to plan long-term actions to maximize amount of pawns agent eats. One has to adjust rewards correctly to allow reinforcement learning system to learn to solve correct task, which can be a challenging task.

2.2 Mathematical framework

Reinforcement learning uses *Markov decision processes (MDP)* [4] as a mathematical framework. More precisely, reinforcement learning tasks that satisfy *Markov property* are called MDPs. A process satisfies Markov property if its possible successive states are only dependent on the current state. A number of variations for MDP exist, but this thesis will focus on *finite-MDPs*. Finite-MDP has finite and countable number of possible states. MDP can also have infinite number of states, e.g. in case of continuous states like object's coordinates in real valued space. Another variation is *partially observable Markov decision processes* (POMDP) (e.g. [28]) which allows states to be partially observed (e.g. in poker, agent can't see opponent's cards). In the case of finite POMDP, one can express the POMDP as a MDP but with uncountable number of states [28].

Finite MDP can be seen as a finite state machine which consists of state nodes S, action nodes A, connections S to A and connections A to S (note how state only leads to new state via an action). Each state $s \in S$ connects to available actions $a \in A$, which then connect to new states which can happen after selecting said action in the given state. Time step t separates states from each other, and successive states have successive time steps. E.g. a_t is the action taken in s_t, and s_{t+1} is the state proceeding the action. Since the process/environment can be stochastic, these connections from actions to states can be non-deterministic. Function $P(s_{t+1} \mid s_t, a_t) \in \mathbb{R}$ determines transition probabilities between states and actions. A *reward function* $R(s_t, a_t, s_{t+1}) \in \mathbb{R}$ [41] assigns reward $r_t \in \mathbb{R}$ to every state transition via an action, which is used to determine which actions are wanted and which are not. Agent will advance in this graph of states and actions in a cycle of receiving a state, executing an action and receiving a reward from the environment. Before agent receives the reward the environment proceeds to the next state which can include interaction of other entities (e.g. other players). This cycle is illustrated in Figure 2.

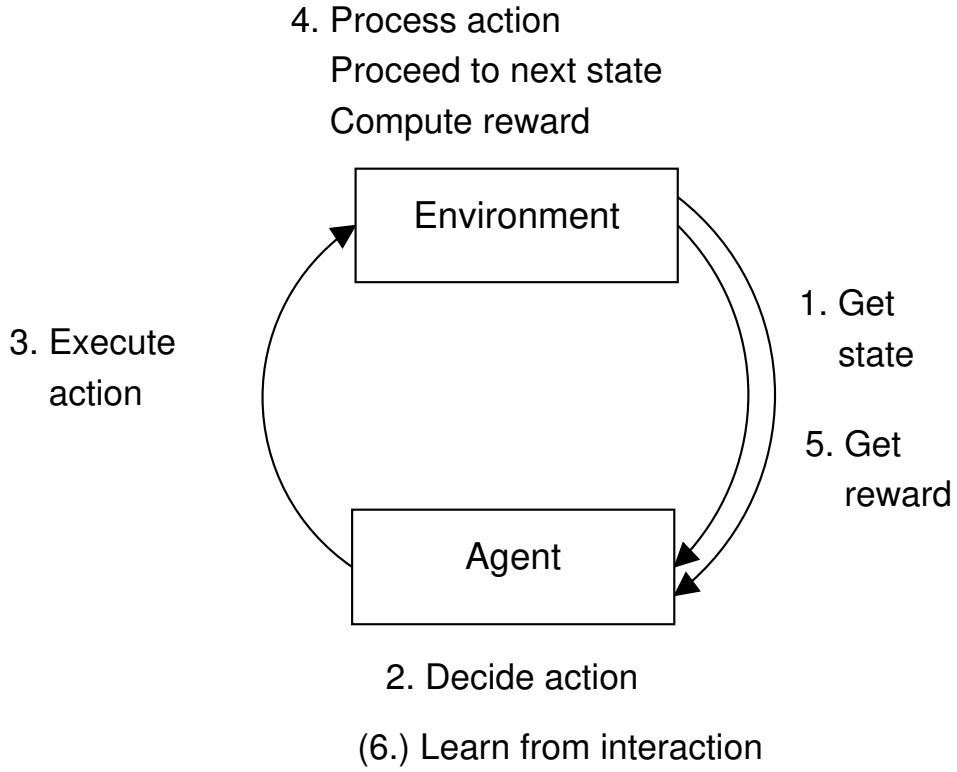


Figure 2: *Interaction step between reinforcement learning agent and environment. This is not a strict model for all reinforcement learning models, e.g. environment can advance multiple steps from one action. Learning can be done after each step or in batches after certain amount of actions.*

For interacting with the environment need to define a *policy* $\pi(s_t)$ [41] which returns an action a_t to execute given a state, defining how the agent behaves in the environment. We try to find the optimal policy $\pi^*(s_t)$ which provides best possible action given any state. Given a policy, we can define *value function* $V_\pi(s_t)$ [41] which returns *value* of a given state. The value describes how good the given state in terms of the given policy and foreseeable future. A similar function is a *state-action value* function $Q(s_t, a_t) \in \mathbb{R}$ [41] that returns value for an action in a given state. Some reinforcement learning systems also utilize a *model* of the system which estimates how MDP will transition from given state and action to a next state (i.e. predict next state given current state and action).

Term "good" is used to describe an action or state that is expected to return a high cumulative sum of rewards in future. A bad action or state would in turn return low cumulative reward. We can also express the "goodness" of a state or action with *value*. If we assume we have optimal policy π^* and correct values for state/state-action value functions, value of state s_t under policy π^* is defined as [41]

$$V_{\pi^*}(s_t) = \mathbb{E}\Big[r_t + \gamma V_{\pi^*}(s_{t+1})\Big]. \tag{1}$$

That is: The value of a given state is expected sum of immediate reward and weighted value of the next state. The state-action value $Q_{\pi^*}(s_t, a_t)$ is defined similarly:

$$Q_{\pi^*}(s_t, a_t) = \mathbb{E}\Big[r_t + \gamma \max_a Q_{\pi^*}(s_{t+1}, a)\Big]. \tag{2}$$

Both (1) and (2) include recursive relationship by using values from successor states, which allows these values to also include information from possible future. In case of state-action value, we optimistically select highest value we could reach from state s_{t+1}.

These equations included a new variable *discount factor* γ [41]. One could count reward from all future states as they are, but we might want to emphasize for higher close-future rewards instead of long-term rewards. With $\gamma = 0$, value of a state only includes the next reward (very short-sighted, akin to a greedy algorithm). With $\gamma \approx 1$, value of a state includes very long term rewards and might choose bad actions (in short term) to reach a very good state in longer term.

While above applies for most states and actions, sometimes there might not always be a next state available. This happens when e.g. game of chess has ended (state s_t was one step away from either player winning). This state without proceeding state is called a *terminal* state, and an *episode* is sequence of actions and states starting from some initial state and ending to a terminal state, e.g. one full game of chess. In practice these terminal states break the bootstrapping, and practical implementations need to treat them appropriately to avoid connecting terminal states to initial state (e.g. player dying suddenly leads to him being alive again, which could be interpreted as a good thing). In case of terminal states, the value function is simply defined

$$V_{\pi^*}(s_T) = \mathbb{E}\Big[r_T\Big]. \tag{3}$$

We use T to represent the terminal time-step, after which no more steps will happen and new episode will start from $t = 0$.

To sum this all up with an example of game of chess: A state s would be one of the pawn setups in the board, an action a would be agent moving a pawn, and transition probability $P(s_{t+1} \mid s_t, a_t)$ models how opponent could react to agent's action. Reward function $R(s_t, a_t, s_{t+1})$ returns positive rewards when an opponent pawn was eaten or game was won, depending on how it is defined. Agent's policy $\pi(s_t)$ provides the (hopefully optimal) actions given a state, and value function $V_\pi(s_t)$ provides estimation on how good the current state is for the agent (e.g. if agent is close to victory the value of such state is high, and if agent is about to lose the value is low). Optional model of this environment would attempt to estimate what action opponent takes given a state and agent's action. A state right before either of players winning the game is the terminal state.

2.3 Value based learning

We can now approach solving our reinforcement learning problem and find the optimal policy. One of the ways to do is by approximating the value and/or state-action value functions and using them to create our policy. This is referred as *action-value methods* [41] (with terms of this thesis, *state-action value methods*) or just *value methods*.

Following methods assume number of states and actions per state are small enough to

form a sensible matrix of all state-action pairs. This helps understanding the methods and is also used in practice in some situations (e.g. chess could be represented with this). For large state or action-spaces, linear approximations of this table may be used. More on this in Chapter 4.

2.3.1 Policy iteration

Policy iteration using dynamic programming is the first method described in [41] Chapter 4. Assuming we know all possible states S, actions A, transition probabilities $P(s_{t+1} \mid s_t, a_t)$ and immediate rewards $R(s_t, a_t, s_{t+1})$, we can find the optimal policy in a finite number of steps. This solution consists of iterations of two phases: Policy evaluation phase calculates new estimate of the value function $V(s_t)$ using the current policy π_t. Then, policy improvement phase uses this new value function to estimate the new version of the policy π_{t+1}. This method converges to π^* when iteration count approaches infinity [41]. We have to repeat this iteration many times, because at first iteration policy only knows states' immediate rewards r_t of states, at second iteration it knows states' value two states ahead of time $r_t + \gamma r_{t+1}$, at third step three steps ahead $r_t + \gamma r_{t+2} + \gamma^2 r_{t+3}$ and so on. The more iterations, the further in future the agent sees when deciding actions. Detailed algorithm is included in Algorithm 1.

A similar approach *value iteration* [41] does not use policy function to update itself, instead operations are truncated in updating value function alone. Instead of taking expected value of future state to be the value of a state, we take maximum of the values of future states plus the immediate reward.

Both of these iterative schemes have been shown to give better policy than the original policy on each iteration unless the original policy is already optimal ([41], pages 89-90).

2.3.2 Monte Carlo and temporal difference learning

One problem of dynamic programming in reinforcement learning is the assumption of complete knowledge of the environment (the model, i.e. transitions $P(s_{t+1} \mid s_t, a_t)$). These are very rarely known in practical applications and thus we can not rely on having this oracle knowledge. Monte Carlo methods for reinforcement learning avoid this issue by using *experiences* instead of explicit model of the environment [41]. An experience e consists of a state, an action, gained reward and a next state. Monte Carlo method runs a large number of games (episodes) in the environment, gathering experi-

Assume known $P(s_{t+1} \mid s_t, a_t)$ and $R(s_t, a_t, s_{t+1})$ for all $s_t \in S$ and $a_t \in A$;
$V(s) \leftarrow$ arbritrary $\mathbb{R}$ for all states;
$\pi(s) \leftarrow$ arbritrary valid action for all states;
$\gamma \leftarrow$ discount factor;
while *Policy function not converged* **do**
  Update value function (policy evaluation);
  while *Changes in value function significant (not converged)* **do**
    for $s_t \in S$ **do**

$$V(s_t) \leftarrow \sum_{s_{t+1}} \overbrace{P(s_{t+1} \mid s_t, \pi(s_t))}^{\text{Weight based on policy}} \overbrace{\Big[R(s_t, \pi(s_t), s_{t+1}) + \gamma V(s_{t+1})\Big]}^{\text{Expected value of going to state } s_{t+1}};$$

    end
  end
  Update policy function (policy iteration);
  for $s_t \in S$ **do**

$$\pi(s_t) \leftarrow \arg\max_a \overbrace{\sum_{s_{t+1}} P(s_{t+1} \mid s_t, a)\Big[R(s_t, a, s_{t+1}) + \gamma V(s_{t+1})\Big]}^{\text{Expected value of taking action } a}$$

  end
end

Algorithm 1: *Pseudo code of policy iteration [41]. By knowing all transition probabilities and immediate rewards, we can update the value function to include future horizon of rewards. We can then use this value function to determine best actions per state. Note that for practical implementation more convergence checks must be added, especially if there are more than one optimal policies.*

ences and then using these experiences to run policy/value iterations instead of a strict knowledge of environment. After an episode we can update our value and/or state-action value functions with the gathered knowledge, since we know future rewards in every state till the terminal state. Essentially we are sampling the environment with a number of episodes.

Even with Monte Carlo approach, policy/value iteration is tedious to update: We require large number of samples and finished episodes before we can run a single update on the policy and the value functions. *Temporal difference* (TD) (Chapter 6. [41]) learning avoids this problem by updating the value/policy function on each time step. With Monte Carlo methods we required to know the return of the whole episode (playing from the start to terminal state and gathering the true sum of rewards), but in temporal difference learning we update the value function with the immediate reward and value of the next state. The sampled immediate reward provides more information about the value of the previous state, while *bootstrapping* to the *known information*

of the value of the next state provides information about future state values. Simplest temporal learning method, TD(0) [41], updates value of a state as follows:

$$V(s_t) = V(s_t) + \eta[r_t + \gamma V(s_{t+1}) - V(s_t)], \tag{4}$$

where η is the learning rate, preferably a small constant $0 < \eta < 1$ [41]. This defines how much of a weight one sample has on our estimate of value function. Without this parameter learning would only focus on the observed experience, instead of recalling the past experiences. While this greedy-looking way of bootstrapping only to the next state first seems suspicious, it has been shown to converge to a optimal policy if learning rate is small enough and tabular value functions are used [41].

Q-learning [49] is a temporal difference method that has very similar structure to (4), only with the action values instead of the state values:

$$Q(s_t, a_t) = Q(s_t, a_t) + \eta[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \tag{5}$$

2.3.3 N-step temporal difference learning

A mix between temporal difference and Monte Carlo methods is to have multiple steps in temporal difference algorithm. The presented Q-learning update (5) includes only one step (next state). We can have the same equation with, say, three steps. We read the states, execute actions and receive rewards of three steps ahead before updating our t state value:

$$Q(s_t, a_t) = Q(s_t, a_t) + \eta \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 \max_a Q(s_{t+3}, a) - Q(s_t, a_t)\right]. \tag{6}$$

This is called n-*step* Q-learning (more generally just n-*step TD* learning) [41]. The discounted sum of successive rewards is called *return*, which here was $G_3 = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}$. With this we may require less visits to state-action pair s_t and a_t to approximate the true value. This is especially useful with large state and action spaces where we might rarely visit a certain state, e.g. reaching winning check-mate in chess.

2.3.4 Eligibility traces

Eligibility traces [41] further generalizes this idea by taking discounted sum of n-step learning values with different n. E.g. With three steps, we would take take discounted sum of 1-step, 2-step and 3-step returns. The λ variable in TD(λ) determines the strength of this discount. Let G_n be return of n steps at some state. The λ-*return* [41] is then defined by

$$r_\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_n\,, \tag{7}$$

where $\lambda \in (0,1)$. This is similar to reward discounting, except we are taking a weighted average over different n-step returns instead of just summing them. With smaller λ the temporally closer states have higher weight. Previously mentioned TD(0) only includes reward from the very next state. TD(1) learning would include all rewards till the terminal state, essentially making it a Monte Carlo learning.

2.3.5 Defining policy function and exploration

So far we have only approximated value and/or state-action value functions but we have not defined our policy function yet. Some of the equations above use *greedy action* [41] by selecting action which has the highest value according to our current state-action value function:

$$\pi_\theta(s_t) = \arg\max_a Q(s_t, a)\,. \tag{8}$$

This method is very intuitive. After all, we wish to maximize our expect reward for the future. However, it has some problems: If we use this policy to select actions while learning, i.e. do *on-policy* learning [41], we might never visit some states. For example: In a game of chess, if our agent eats enemy pawn with the king it will experience that action as a good experience. However, after this parameter update it will always use king to eat the enemy pawn in that same situation, and will not try any other actions in that state. This is problem of *exploitation vs. exploration* [41]: How we should balance between exploiting current knowledge and exploring new, unseen actions?

One simple approach to this is to use ε-*greedy policy* [41]. Instead of always choosing

the action with highest expected reward, we choose a random action at some probability ε:

$$\pi_\theta(s_t) = \begin{cases} \arg\max_a Q(s_t, a) & \text{at probability } 1-\varepsilon \\ \text{random available } a & \text{at probability } \varepsilon \end{cases}. \tag{9}$$

This way we have the advantage of not getting with same action every time. Another exploration technique is to use *exploring starts* [41], where each episode starts from a random state. However, this is not always practical (e.g. You only have one initial state in a game of chess).

Using ε-greedy policy has very short sighted exploring. If certain sequence of actions is required to reach reward, this random action selection may take very long to randomly pick this sequence. In some cases some actions might even cancel each other out. In VizDoom [22] agent can chose to turn left or right certain amount of degrees. If policy randomly alternates between these two, these turning actions will cancel each other out and agent does not really move anywhere.

More sophisticated exploration methods exist (e.g. [32]), but these are outside of the scope of this thesis. Following policy based methods partly avoid this issue by having stochastic policy, randomizing the actions we take automatically.

2.4 Policy based learning and actor-critic

Previously described value based learning does not explicitly optimize policy itself and requires an implementation for policy using state-action function (e.g. greedy, ε-greedy). It also might be computationally infeasible to implement state-action pairs, even via a function approximator like a neural network if the number of actions and states is too large. In the case of continuous actions we can not estimate values for each possible action without some ad-hoc solutions like quantizing .

Policy approximation [41] avoids this problem by attempting to learn policy function itself. One way to do this is to use *actor-critic methods* [41]. An actor-critic system has two parts: A critic and an actor. The actor (policy) chooses which actions to take and critic gives reward based on how good the action was. We have two separate sets

of parameters, one for actor and one for critic, which we optimize at the same time. Our policy estimator, the critic, now outputs a distribution over actions instead of one explicit action.

We can train our critic with value based learning methods like above, like TD(0). For actor we simply need to increase or decrease the probability of the taken action based on critic's output. One simple way is to use TD error [41]

$$c = r_t + \gamma V(s_{t+1}) - V(s_t)\,, \tag{10}$$

where $V(s)$ is the critic / value function we also approximate. If our policy is a table of state-action pairs, we can update the critic

$$\pi(s_t, a_t) = \pi(s_t, a_t) + \eta c\,, \tag{11}$$

where η is the learning rate and policy $\pi(s_t, a_t)$ returns a probability of given action at given state. Note that output of the policy should be a distribution which requires additional calculations in this case, but this is only one of the ways to implement actor-critic system. *Policy gradients* [42] use gradients of the expected/average reward function to know which way policy's parameters should move to increase the expected reward, which is very similar to gradient descent used for machine learning (see Chapter 3.2).

2.5 Evaluating reinforcement learning methods

Normally in machine learning (and other prediction, pattern recognition and similar tasks) running experiments have at least two distinct parts [15]: Training and evaluation of the model. In a classification task, training phase consists of optimizing parameters of the model with a given training set. Evaluation phase then uses separate samples/data to test how good the trained model is at classifying new, unseen samples. For reinforcement learning such datasets rarely if ever exists. Some tasks would require a very large number of stored states/actions to define a reasonable dataset, and the general idea of actions leading to different states may require observing all the 'paths' states can go. Instead of datasets we can use environments and rules which set a baseline for comparing systems. For example, ViZDoom [22] defines set of environ-

ments and reward functions with different tasks one can use.

Reinforcement learning systems are often evaluated on how well they perform in a given task and on how much computational resources it took to reach such performance. Especially figures plotting time trained versus performance are often shown in studies (e.g. [26, 22] to visualize how performance increases during training. With most recent studies and advances in the field, some studies compare their systems against humans (e.g. [27, 39]) to see if we can obtain so called "super-human" performance on a given tasks. While analyzing the performance, one should note that **reward function may not represent ultimate goal of a video game or task**. Reward function can be coded to reward for winning game, but also to reward from intermediate steps like killing one enemy.

If provided environment is a complex one (e.g. video game, physical world), separate episodes may include completely different set of observed states and available actions. A game of chess always has the same starting position but unknown actions of the opponent player are likely to create different path of states for each match. This variance can be mitigated by taking an average of measurements over multiple training and evaluation runs with new set of parameters, since training procedure can be affected by the same stochastic behavior of the environment and model's initial parameters. Examples of reward vs. training graphs can be seen in Figure 3, Figure 2 in [27] and Figure 1 in [26]. Note the number of experiments ran to obtain one curve used for comparing, e.g. "average over the best 5 models from 50 experiments" [26].

A common problem of overfitting might also exhibit in reinforcement learning systems. However, since states, actions and task(s) might not be easily separable the overlap between training and evaluation scenarios might be hard to detect, even with proper domain knowledge. Depending on the research question this is not a problem (e.g. overfitting to the provided scenario is exactly what we are after), but if one wants to train more generalized system the simplest path is to provide more different training scenarios and switching between them during training.

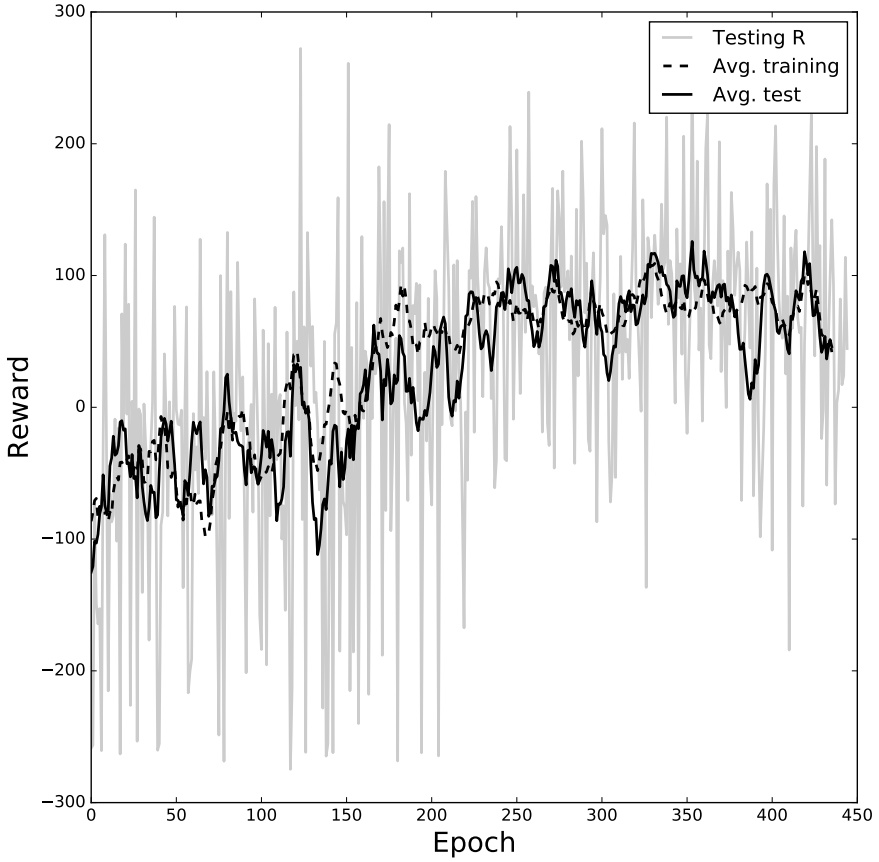


Figure 3: *Plot of a reinforcement-learning agent learning to play Doom via ViZDoom [22]. Reward of individual episodes (epoch) varies considerably and even a moving average with window of* 10 *epochs produces a noisy signal. Multiple evaluation runs, or even training runs, are required for confident evaluation score.*

3 Neural networks and deep learning

To understand many of the methods and challenges of reinforcement learning it is good to know some general machine learning (or pattern recognition) terminology and methods. While machine learning can be used to solve number of different types of tasks in practice they all boil down to finding dependencies, correlations and connections in the provided data, inferring or predicting new information from the given input.

Machine learning can be divided into different groups by different metrics, but one way to split them is by their *task*. This list of tasks [15] has high number of items but we focus on two common ones: Regression and classification. Regression task aims to predict a real-valued scalar from given input, while classification focuses on finding a correct label for the input (select one label from given set of labels). These two are not the only major topics in machine learning but this thesis will focus on them as an basic example of what machine learning can do. Later we will combine these with reinforcement learning in Chapter 4.

Generic structure of machine learning and pattern recognition is to gather data we want to predict or analyze, build a model and a loss function around this data and then train model's parameters using the gathered data [15]. Models often include unknown parameters and weights which allow them to adjust to different types of data, but this requires finding optimal parameters for this specific data in question.

While the general idea of finding model that predicts the data as accurately as possible sounds similar to a look-up table or a recall task, there is one distinct difference: Machine learning aims to *generalize* on the data, meaning it tries to produce accurate outputs for inputs it was not trained for [15]. Since physical world measurements always include noise and unwanted interference, correct machine learning model can ignore this noise component while finding the generalized solution for the samples.

3.1 Model

As machine learning and pattern recognition can be used for several different tasks, naturally there are several different models and approaches included in this area. E.g. Classification can use probabilities and terms like "Sigmoid function" and "softmax"

are common, but regression often wants other values than just $[0, 1]$, and sometimes multiple outputs are required (e.g. noise removal, sequence-to-sequence prediction). Not only tasks differ but the structure of available data differs: We might only have few observations with few dimensions, or millions of samples with few dimensions or millions of samples with thousands of dimensions, the underlying manifold of the data might be linear or something more complicated, there might or might not be strong correlations or even linear dependencies between components and so on. This thesis will only mention fraction of available models with focus on what has been used along with reinforcement learning.

3.1.1 Starting simple: Linear and logistic regression

One of the models is the linear regression model [6] that fits a straight line $\hat{y} = \boldsymbol{w}_1 + \boldsymbol{w}_2 x$, where x is an input variable, $\boldsymbol{w}_2$ is the weight for input variable, $\boldsymbol{w}_1$ bias term (or intercept term) and $\hat{y}$ is the estimated output. An extension to this is a polynomial regression which allows fitting different order model in to the data, e.g. second order model would be $\hat{y} = \boldsymbol{w}_1 + \boldsymbol{w}_2 x + \boldsymbol{w}_3 x^2$. This transformation on vector of inputs $\boldsymbol{x}$ is called *feature mapping* [6], where we can also use different transformations (e.g. different powers, logarithms). We can also have multiple input variables, and with matrix multiplication we can express it as

$$\hat{y} = b + \boldsymbol{w}^T \boldsymbol{x} \,. \tag{12}$$

$\boldsymbol{x} \in \mathbb{R}^N$ is now column vector of inputs with N elements, and $\boldsymbol{w} \in \mathbb{R}^N$ is a column vector of weights. The term b is called an intercept or a bias term which offsets the fitted curve. Without this term the fitted line would always pass through origin, which would limit model's ability to model some data. One can also include b in the weights w by appending a constant 1 to the input vector, so we can write (12) as $\hat{y} = \boldsymbol{w}^T \boldsymbol{x}$. Note that now $\boldsymbol{x}, \boldsymbol{w} \in \mathbb{R}^{N+1}$.

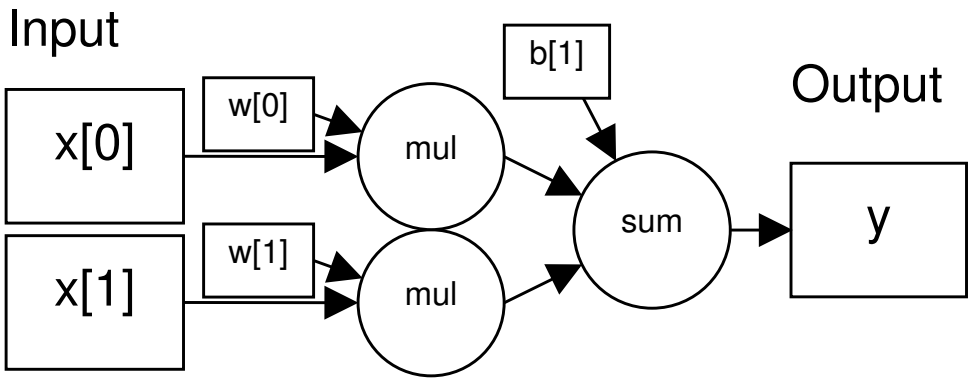
Regression model [6] can also be transformed to suit classification tasks: *logistic regression* is defined by

$$\hat{y} = \frac{1}{1 + e^{-(b + \boldsymbol{w}^T \boldsymbol{x})}} = \sigma(b + \boldsymbol{w}^T \boldsymbol{x}) \,, \tag{13}$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the *logistic function* [6]. Logistic function limits the value of $\hat{y}$ in interval $(0, 1)$ which can be interpreted as a probability for a classification task. Without further modifications this can be seen as fitting a hyper-plane in the data that separates two classes. Note that while the model seems very similar to normal regression it requires different optimization techniques.

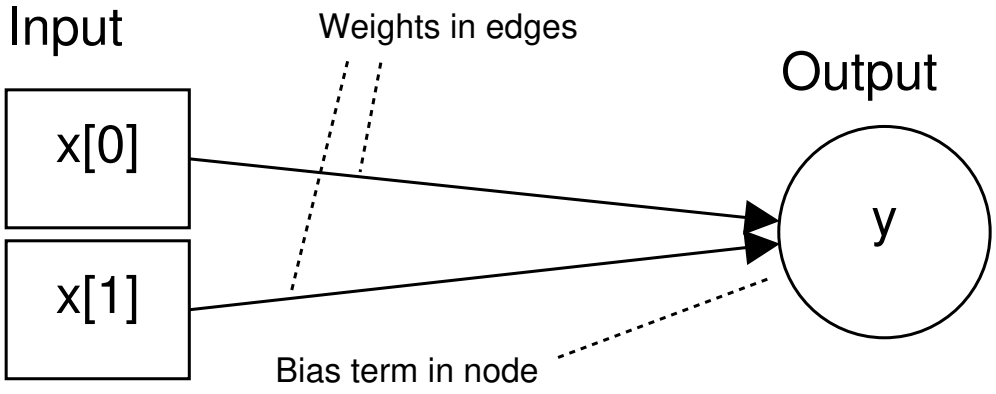
3.1.2 Towards neural networks

With these two models we can already get to *artificial neural networks* [6] (or just "neural networks"). Neural networks can be thought as a number of *neurons* (or *nodes*, *units*) connected to each other to form an output, but one can also think it as an extension to previously seen regressions: Lets take a single linear regression that takes column vector $\boldsymbol{x} \in \mathbb{R}^2$ with two variables as an input. We can visualize it as a computational graph:



The computational process starts from left (input) and ends to output. Squares are variables or inputs, circles are operations executed and arrows indicate inputs for operations.

To simplify our future graphs let each edge include a weight parameter w and each node include a bias term b. A node then sums up inputs and bias term, weighting inputs by weights in their respective edges. Note that bias term is not a single vector, but we separate constants from each other with subscript. Our linear regression can then be visualized as:

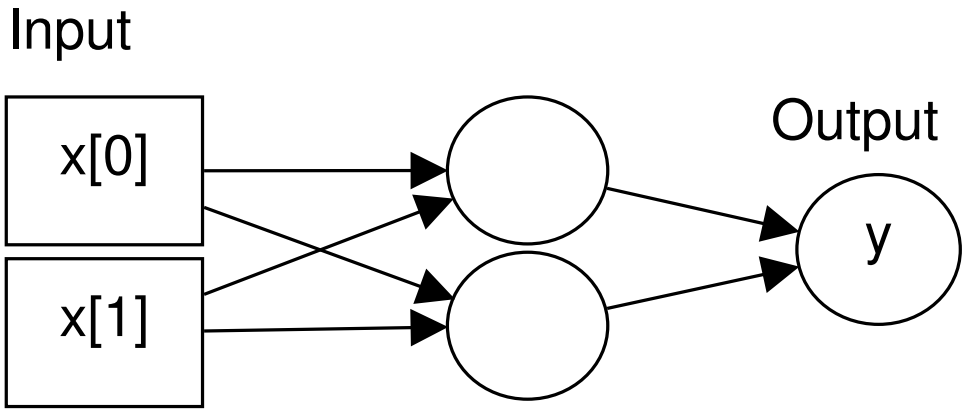


We can fit multiple linear regressions with same inputs, weight their outputs and sum them together. Let a_1 and a_2 be coefficients of the linear combination of the linear regressions, and we get:

$$\hat{y} = b_3 + a_1(b_1 + \boldsymbol{w}_1^T \boldsymbol{x}) + a_2(b_2 + \boldsymbol{w}_2^T \boldsymbol{x})\,. \tag{14}$$

Here $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are weight vectors for the two nodes, separated by the subscript number. The bias terms are also scalars, also separated by the subscript.

As for the simplified computational graph we get:



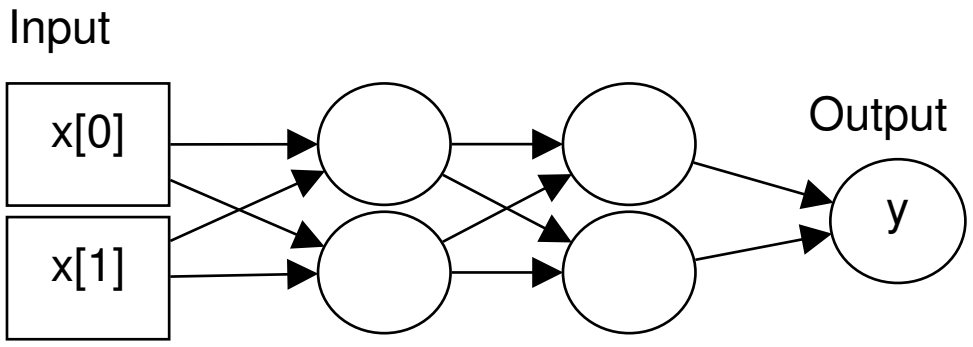
While this might first seem beneficial for a regression task, it will still end up being a linear combination of inputs, which will be shown later. We will later add non-linear functions after dot products to approximate wide variety of functions.

We can also add more nodes in terms of "depth": We feed the output of our linear regression models to yet another linear regression. Let $o_1 = \boldsymbol{w}_1^T \boldsymbol{x}$ and $o_2 = \boldsymbol{w}_2^T \boldsymbol{x}$

(outputs of the middle nodes in last graph), and thus we can write our two-layer model as

$$\hat{y} = b_5 + a_1(b_3 + \boldsymbol{w}_{31}o_1 + \boldsymbol{w}_{32}o_2) + a_2(b_4 + \boldsymbol{w}_{41}o_1 + \boldsymbol{w}_{42}o_2)\,, \tag{15}$$

where $\boldsymbol{w}_{31}$ is the first element of vector $\boldsymbol{w}_3$ and so on. For the graph we get



This is the basic structure of the neural networks: You have one or multiple levels of *hidden layers*, each of which contain one or more nodes which do a linear combination of the outputs from the previous layer. The weights used in the linear combination are unknown parameters, which will be modified to solve the task (see Chapter 3.2)

However, as this is a essentially a linear combination of linear combinations and the result is still a larger linear combination (with an added bias): Let $\boldsymbol{x} \in \mathbb{R}^n$ be the input features, $\boldsymbol{W}_0 \in \mathbb{R}^{m\times n}$ and $\boldsymbol{b}_0 \in \mathbb{R}^m$ be weights and biasses of the first layer and $\boldsymbol{W}_1 \in \mathbb{R}^{k\times m}$ and $\boldsymbol{b}_1 \in \mathbb{R}^k$ weights and biasses of the second layer. The output of the second layer is then:

$$\begin{aligned}
&\boldsymbol{b}_1 + \boldsymbol{W}_1(\boldsymbol{b}_0 + \boldsymbol{W}_0\boldsymbol{x}) && \mid \boldsymbol{A}(\boldsymbol{B}+\boldsymbol{C}) = \boldsymbol{AB} + \boldsymbol{AC} \\
&= \boldsymbol{b}_1 + \boldsymbol{W}_1\boldsymbol{b}_0 + \boldsymbol{W}_1\boldsymbol{W}_0\boldsymbol{x} && \mid \boldsymbol{W}_2 = \boldsymbol{W}_1\boldsymbol{W}_0,\ \boldsymbol{b}_2 = \boldsymbol{b}_1 + \boldsymbol{W}_1\boldsymbol{b}_0 \\
&= \boldsymbol{b}_2 + \boldsymbol{W}_2\boldsymbol{x}\,,
\end{aligned} \tag{16}$$

where $\boldsymbol{b}_2 = \boldsymbol{b}_1 + \boldsymbol{W}_1\boldsymbol{b}_0 \in \mathbb{R}^k$ and $\boldsymbol{W}_2 = \boldsymbol{W}_1\boldsymbol{W}_0 \in \mathbb{R}^{k\times n}$. Since we end up with the same equation we started with, we can apply it recursively to any number of fully connected layers.

Thus output of a neural network like this is linear combination of inputs. Such neural

network can not, for example, learn to predict XOR function with this setup due to XOR's requirement for non-linearity. Imagine plot where x-axis and y-axis are the inputs to a XOR function, outputs 1 are the first class and outputs 2 the second class. You can not draw a single, straight line separating these two classes from each other. You need a curving line to be able to do it, i.e. requires non-linearity.

3.1.3 Activation functions

To address this issue we can include non-linearity to the network with *activation functions* (or *non-linearity functions*) [6]. An activation function $g(\cdot)$ transforms the input in some non-linear way. In the past, logistic function $\sigma(x)$ and similar hyperbolic tangent ("tanh") $\tanh(x)$ has been used as the activation function. Logistic function outputs values in $(0, 1)$, making it a suitable function for binary classification and producing probabilities (this was also used in logistic regression earlier). However, gradients of these functions saturate (approach zero) when input either increases or decreases. For this reasons using learning with gradients is slow with logistic units, especially through multiple layers of logistic units [15].

In modern neural networks, *rectified linear units* (ReLU) $g(x) = \max(0, x)$ have been used with increased performance. ReLU activation behaves well during optimization because of its linear nature [15] while still allowing modeling non-linear functions when number of nodes his high. ReLU also has convenient gradients of zero when unit did not active (i.e. $x < 0$) and one when it did activate. However, when unit does not activate no learning will happen as we do not have direction for lower error. This can result to dead nodes that never activate. A simple modification to cope with this is *leaky ReLU* $g(x) = \max(\alpha, x)$ where α can be a small constant or even a learn-able parameter (*Parametric Rectified Linear Unit*, PReLU [17]) . Leaky ReLU prevents creation of dead nodes and guarantees non-zero gradient everywhere.

Output layer of the network uses activation function most suitable for the task at hand. For regression we might want to have unbounded values, the whole $\mathbb{R}$, thus we use linear units. In binary classification tasks a logistic unit allows interpreting the output as a probability. If we have multi-class classification, we can use *softmax* function [6] which provides a probability distribution over the output variables. Let $\boldsymbol{x} \in \mathbb{R}^N$, the

softmax function is defined as

$$\text{softmax}(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_{j=1}^{N} e^{\boldsymbol{x}_j}}, \tag{17}$$

after which summing over $\sum_i \text{softmax}(\boldsymbol{x})_i$ will result to one. This is similar to e.g. normalizing vector to unit length, but with exponent specifically selected to be e.

Nodes in the network do not need to share same activation function. In addition to final layer we can have e.g. logistic functions in between ReLU functions to limit the interval of values or just use linear units to have simple weighting and possibly faster training due to simpler functions. We can even have softmax layers in between hidden nodes, intuitively turning values flowing through network into proper probabilities that sum to one. A large number of activation functions exist, but according to [15] many differentiable functions perform well for this task and researchers try out different functions for their research to see which works best.

Combining all these ingredients we can now express a single layer of a network with m nodes. Let $\boldsymbol{o}_i \in \mathbb{R}^k$ be outputs of the previous nodes (or inputs), $\boldsymbol{W} \in \mathbb{R}^{m \times k}$ a matrix of weights, $\boldsymbol{b} \in \mathbb{R}^m$ a vector of bias terms and $g(\cdot)$ the activation function. Output of our layer $i + 1$ is then

$$\boldsymbol{o}_{i+1} = g(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{o}_i) \in \mathbb{R}^m . \tag{18}$$

This is often called a *fully connected layer* because all nodes of previous layer are connected to all nodes of next layer. Layers do not specifically have to be fully connected, and in some cases it is beneficial to use e.g. *skip connections* [15] where multiple previous layers connect directly to one layer, not just the previous layer. These type of networks where values only flow to one direction is also called *feed-forward network* [6]. We can also have loops in the graph, creating so called *recurrent neural network* (RNN) [15].

3.1.4 Recurrent neural networks

Fully connected layers can handle data that has fixed size, but some data like natural language or audio has a varying sequence of elements along with some temporal de-

pendency between elements. The data could be limited to certain size and/or padded with zeros to match the fixed size of the network, but more elegant solutions exist, such as *recurrent neural networks* [15].

Recurrent neural networks are a family of network architectures with loop-like connections in their computational graph. Having loops in the computational graph allows neural network to have an inner state based on previous inputs which can be seen as a type of memory between network activations. A recurrent network would thus produce a *hidden state* $\boldsymbol{h}$ in addition to regular when given parameter $\boldsymbol{x}$:

$$\boldsymbol{h}_{t+1}, \hat{y}_t = f(\boldsymbol{h}_t, \boldsymbol{x}_t \mid \theta) \ . \tag{19}$$

With the recursive structure we can see the modeling of the dependency between elements in a sequence, as the hidden state (and thus output of the network) is dependent on all inputs and states previous to it. In practice we unfold this function only by set amount to control the computational cost.

Recurrent neural networks can create hidden states by number of ways, e.g. by having nodes that connect to themselves or outputs connecting directly to the input. With a layer of nodes which connect to themselves exist in a network, its output can be defined as

$$\boldsymbol{y}, \hat{h}_t = g(\boldsymbol{b} + \boldsymbol{W}_h \boldsymbol{h}_{t-1} + \boldsymbol{W}_x \boldsymbol{x}) \ , \tag{20}$$

where $\boldsymbol{x}$ is input or output of last layer, $\boldsymbol{W}_h$ weights for hidden states, $\boldsymbol{w}_o$ weights for inputs, $\boldsymbol{b}$ biasses and $g(\cdot)$ activation unit. This network can be then unrolled in time by a set amount, and then use regular back-propagation algorithm (see Chapter 3.2). This is called *back-propagation through time* [15].

However, unrolling the network and multiplying values multiple times with the same weight matrix $\boldsymbol{W}_h$ causes the norm of the gradient to decrease or increase, same as multiplying value with many < 1 values or > 1 values. Most of the time gradient will vanish (approach zero) but it can also sometimes explode (approach infinity), both of which will hurt the optimization [15]. For this reason other types of RNN methods are required for efficient training. Currently *gated RNNs* [15] like *Long-Short term mem-*

Name	Definition	Description
Linear	$g(x) = x$	In $(-\infty, +\infty)$. Does not change the output or add non-linearity. Can be used as a activation function of the final layer for an unbounded regression output.
Sigmoid	$g(x) = \frac{1}{1+e^{-x}}$	In $(0, 1)$. Commonly used in the past with success but can not be used with larger nets due to saturation if x is too positive or negative (derivate approaches zero).
Tanh	$g(x) = \tanh(x)$	In $(-1, 1)$. Very similar to Sigmoid unit with same saturation problems. Generally performs better than sigmoid units in neural networks [15].
ReLU	$g(x) = \max(0, x)$	Rectified linear unit. In $(0, +\infty)$. Common in deep learning and modern nets due to performance vs. Sigmoid/Tanh unit. Can have "dead" nodes if $x < 0$ for all inputs (always outputs 0).
Softmax	$g(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$	In $(0, 1)$ and $g(x_i)$ sums to 1. Often used as a final layer to produce probabilities, can also be used in hidden layers.

Table 1: *List of common non-linearity functions (or activation functions) used with neural networks.*

ory (LSTM) and *gated recurrent units* are used for sequential data modeling. These may also allow having input sequences of varying size.

3.2 Optimizing model parameters

Models alone often is not enough to do machine learning due to unknown parameters which are required to model work correctly. A set of models exist which do not have parameters modeling the data but rather control the complexity of the model, called *nonparametric methods* [6]. These can also exhibit high performance but also have some drawbacks, e.g. *K-Nearest Neighbors* (KNN) [6] does not require any training phase but requires storing all the training data, which can be very space consuming and computationally expensive later on (optimization structures can be used though).

3.2.1 Training and testing

Finding the correct parameters or good approximations is called *training phase* or *learning phase* [6] of the model which uses *training dataset* [6]. This training dataset is a set of observations (feature vectors) of event we want to model, and is accompanied by *training labels* (also called *targets* or *ground truth*). In such case the machine learning is of *supervised learning* where we know what our outputs $\boldsymbol{y}$ should correspond when given inputs $\boldsymbol{X}$. The goal is to find parameters θ so that our model's estimations $\hat{\boldsymbol{y}}$ are as close as possible to labels $\boldsymbol{y}$. Additionally, the point is not to make perfect recall of samples and remember which output corresponds to which input, instead we try to *generalize* learned information from training data to unseen samples [6]. For example, learning individually $a + b = c \quad a, b, c \in \mathbb{N}$ could require large amount of memory (e.g. a look-up table). Instead we learn the abstract concept of summing natural numbers, which we can apply for any set natural numbers without knowing beforehand the results.

To know how well our model truly performs to unseen data, we also have *test dataset* and *test labels* (or *evaluation set*) which does not overlap with the training dataset. Point is to find approximated parameters with training dataset and then try trained parameters on test dataset to see model's performance against unseen data. Test dataset should represent real situation where system is used, so **test data should now be used to tweak model parameters** because we can not do that during practical use of the system. Sometimes third dataset called *validation set* is used for tuning hyper-parameters of the system and attempting different training methods. None of the design decision should be based on the results obtained from test set, otherwise there is a chance of *overfitting* to the obtained data (more in Chapter 3.4).

Correct parameters such that $\hat{\boldsymbol{y}} = \boldsymbol{y}$) rarely exist. For example, linear regression model can not have 100% accuracy in data where samples form something else than solid line. Because of this we need to find best compromise between results, which can even be better than finding 100% accuracy in the training set (see Chapter 3.4).

3.2.2 Loss function

To find good parameters for our task we need a way to compare different parameters against each other in terms of their performance, so we define a *loss function* $\mathcal{L}$ [6] (also *cost function*, or the opposite *objective function*). Loss function returns a scalar

representing how bad the current model with current parameters is at predicting labels, and thus now our goal is to minimize loss:

$$\arg\min_{\theta} \mathcal{L}(\boldsymbol{X}, \boldsymbol{y} \mid \theta) \,. \tag{21}$$

For a practical example lets look at the simple linear regression. We can use *least squares* loss function, which attempts to minimize squared error between predicted and true outputs. Least squares is defined by

$$\mathcal{L} = \sum_{i=1}^{N} (\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)^2 \,, \tag{22}$$

which can also be written in matrix form $\mathcal{L} = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|^2$. When $\hat{\boldsymbol{y}}$ matches exactly $\boldsymbol{y}$ the loss will be zero. To minimize this loss we take a partial derivative with respect to weights $\boldsymbol{w}$ and finding where slope is zero, which corresponds to minimum of the loss function. Finally we end up to solution a closed form solution [6]

$$\boldsymbol{w} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X} \boldsymbol{y} \,, \tag{23}$$

thus we can directly compute the weights that produce least squared error between estimates. This same equation also applies to polynomial regression where we replace feature vectors $\boldsymbol{X}$ with modified vectors that include cross-products and higher order transforms.

However, this is one of the rare cases where optimization is this easy, and more than often we will not have closed form solution (i.e. we can not compute it in finite steps on a computer). This is especially case with the neural networks and their activation functions.

3.2.3 Gradient descent

This is where we get to *iterative methods* of optimization, namely *gradient descent* [6] (also called "*gradient search*"). The core idea of gradient descent is to use the gradient of the loss to find the direction towards smaller loss, and taking small steps towards that direction. Let $\boldsymbol{x} \in \mathbb{R}^N$ be a vector of variables and $f(\cdot)$ function (e.g. loss function).

The gradient of $f(\boldsymbol{x})$ w.r.t $\boldsymbol{x}$ is defined as:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \left(\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}_1}, \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}_2} \cdots \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}_N} \right)^T . \tag{24}$$

When calculating gradient of a loss we usually calculate gradient w.r.t model parameters θ. This way we know which way each parameter should move to decrease (or increase) the loss.

An extension to this is using second-order partial derivatives, creating a Hessian matrix. Hessian is calculated by taking gradients of first-degree gradient. Using the definitions from earlier equation we can compute the Hessian matrix $\boldsymbol{H}(f)(\boldsymbol{X})$ (notation $\nabla\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ is also used) [6]

$$\boldsymbol{H}(f)(\boldsymbol{X}) = \begin{pmatrix} \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}_1 \partial \boldsymbol{x}_1} & \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}_1 \partial \boldsymbol{x}_2} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}_1 \partial \boldsymbol{x}_N} \\ \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}_2 \partial \boldsymbol{x}_1} & \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}_2 \partial \boldsymbol{x}_2} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}_2 \partial \boldsymbol{x}_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}_N \partial \boldsymbol{x}_1} & \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}_N \partial \boldsymbol{x}_2} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial \boldsymbol{x}_N \partial \boldsymbol{x}_N} \end{pmatrix} . \tag{25}$$

Using Hessian along with gradient is same as taking Taylor series with two degrees, creating more accurate representation of the points around $\boldsymbol{x}$ and thus allows taking more accurate steps towards minimums. Hessian matrix requires squared amount of space with regards to parameters in the function. In case of neural network loss function this requirement can be too high to be computationally feasible [6]. Despite this it sometimes used in optimization tasks (e.g. Newton's method uses Hessian), and a number of approximations with lower time complexity exist that can be used with neural networks [6].

Minimizing the loss means we want to find the minimum of the loss function, and this can be done with gradient by taking small steps towards the decreasing direction. For this we have to take the gradient with respect to the parameters θ, basically computing a vector with a derivative for each parameter. The gradient descent itself is defined as

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\boldsymbol{X}, \boldsymbol{y} \mid \theta) , \tag{26}$$

where θ is the model's parameters, and η is a small constant controlling the step size we take. Too large and we may oscillate around minimum, too small and we may take unnecessary long to reach our goal. We subtract the gradient from parameters as derivative tells which way loss increases.

Because single data-point rarely represents the distribution of the data, we have to include multiple sample points from training data while calculating gradient to reduce variance between parameter updates:

$$\theta_{t+1} = \theta_t - \eta \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \mathcal{L}(\boldsymbol{X}_i, \boldsymbol{y}_i \mid \theta)\,. \tag{27}$$

This is called *batch-learning* [6] as we go through all the samples to compute the loss.

One can continue iterations fixed amount of steps, or until the loss does not change enough. A good thing about gradient descent is that one only needs the gradient of the loss to start optimizing weights, thus you can apply it to many different models and loss functions. Figure 4 illustrates gradient descent of a one-dimensional loss function with two different learning rates.

3.2.4 Back-propagation and the chain-rule

Since neural networks consist of multiple layers of multiple smaller functions, optimizing the parameters is not as straight forward, but we can still use e.g. the gradient descent method described earlier. Major difference is computing the gradient of the weights/parameters, but it can be effectively solved by using high-school math *chain-rule* [6] for computing derivatives of functions composed of two functions. We can apply chain-rule along with sum-rule and product-rule of calculus over again to decompose all parameters in network into trivial mathematical operations.

This procedure of flowing information from cost, computed by a *forward pass*, and computing the gradient for each parameter is called *back-propagation algorithm* [9]. Note that only calculation of gradients is included in back-propagation algorithm, the update of parameters using these gradients is explained in following chapters.

Chain-rule allows us to split a more complicated equation into smaller pieces, differentiating those and combining the results by use of the chain-rule. Let $f(x)$ and $g(x)$ be some functions and $y(x) = f(g(x))$. Derivative for y with chain rule is then defined

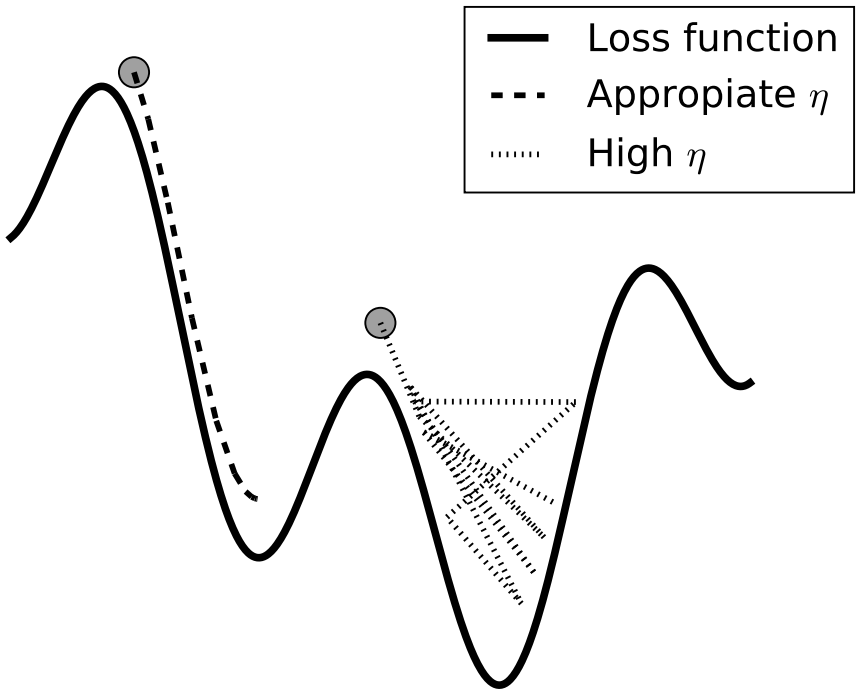


Figure 4: *An illustration of the gradient descent where the loss function is one-dimensional. Two gradient descents were started from gray dots, proceeding along the dashed and dotted lines. Dashed (left) has an appropriate learning rate and steadily descents into a local minimum. Dotted line (right) has too high of a learning rate, causing it to jump back and worth and not descending into the pit. Dashed line gets stuck in a local minimum which is not a global minimum (next minimum).*

as

$$y'(x) = f'(g(x)) \cdot g'(x)\,, \tag{28}$$

which is same as

$$\frac{dy}{dx} = \frac{dy}{dg(x)} \cdot \frac{dg}{dx}\,. \tag{29}$$

By applying chain-rule recursively we can break down an arbitrary equation into trivial mathematical operations which have trivial derivatives, and then combine these parts back into derivative of the original equation. However, when applying this approach to a more complex equation (say, neural network) this tends to repeat some expressions multiple times and especially in larger equations can result to exponentially expensive to compute with respect to number of expressions [15]. Python Theano library [43] uses this approach while calculating gradient but with a large number of optimizations to make computing such graph feasible. Theano (and TensorFlow [1]) create an additional graph for computed gradient, making it *symbolic differentiation*, which automatically gives us gradients for each parameter and results to a new computational graph that can be handled as a regular graph (we can take another gradient et cetera).

3.3 Parameter updates

With calculated gradient we can apply update rule like gradient descent (see equation (26)). This works even with larger neural networks, until number of observations in your data starts getting very high, which is required for e.g. high dimensional data, sparse data and/or large neural networks. Since we might require millions of steps to reach good minimum in loss, we would like to keep gradient computation relatively fast. Unfortunately with increasing number of samples one iteration can take too long as we need to loop over whole dataset always, and whole computation slows down. Luckily there is an simple, but effective, solution for this: *Stochastic gradient descent* (SGD). Instead of using all of the feature vectors of $\boldsymbol{X}$, select one randomly per iteration and do the weight update. With this the computation of loss is constant w.r.t number of samples but converge requires more steps. However, this allows monitor-

ing learning without long waits between iterations and it is still able to converge to a minimum [15]. SGD does not have to take only one sample at a time, instead it can take a *mini-batch* of K samples to average over. This way each the update is a rough estimation of using whole population and can be parallelized during update step.

3.3.1 Momentum

While theoretically sound, SGD can be slow to converge to the optimal solution due to small learning rate. One solution for this is to use *momentum* [33] to update the parameters. Similar to position and velocity in physics, we update the exponentially decaying momentum (velocity) of the parameter updates and use said momentum to update the actual parameters. This speeds up the convergence but can overshoot the optimal point due to momentum. *Nesterov momentum* [40] is an improved version of momentum with lower theoretical upper bound for error, where the parameters are changed twice per update: First based on the momentum and then based on normal gradient in that new spot. However, Nesterov momentum does not improve convergence rate in stochastic learning [15]. Momentum and Nesterov-momentum algorithms are detailed in Table 2.

3.3.2 AdaGrad

Even with momentum we still have to select learning rate. The learning rate can be annealed from a higher value closer to zero to start off with faster moving and the gradually getting more precise during training process. However, we can also dynamically adjust the learning rate per step to an appropriate value based on previous gradients and value sizes. AdaGrad [10] algorithm accumulates sum of squared gradients of previous updates per parameter. The step size per parameter is then calculated by dividing learning rate by square root of the accumulator value. Essentially, parameters with large gradients have smaller step sizes and vice versa, balancing the velocity in parameter space between parameters. This convergences faster in low-gradient areas but the initial accumulation of gradients can throw parameters away from optimal point and hinder the whole training process [15]. Algorithm of AdaGrad is shown in Table 2.

3.3.3 RMSProp

Similar to AdaGrad, RMSProp [44] accumulates sum of squared gradients per parameter, but also includes a exponentially decaying factor for the accumulator. Essentially

this limits knowledge of the gradient sizes to the more local area of parameter space instead of accounting also the gradients at the very beginning of the training. Empirically this has been shown to be perform better than AdaGrad [15], and is one of the common update rules used in deep learning related research [15]. Algorithm for RMSProp is shown in Table 2.

3.3.4 AdaDelta

Another modification to AdaGrad is AdaDelta [51] algorithm. In addition to accumulating gradients, AdaDelta also accumulates squares of updates. Both accumulators also accumulate mean square instead of just squared values. Final learning rate is then ratio between accumulated root mean square of updates and root mean square of gradients. This method allows preserving information about unitness of the parameters by Hessian approximation, which is expected to improve performance ([51], Chapter 3.2). Empirically AdaDelta performs better than AdaGrad [51]. Interestingly this algorithm only requires global learning for the first, unlike AdaGrad and RMSProp who use global learning rate for all updates. Algorithm for AdaDelta is shown in Table 2.

3.3.5 Adam

Adam [23] algorithm combines the benefits for these accumulators and momentum. Adam accumulates first and second-order (power of two) momentums of the gradients, does exponential decay to discard temporally distant information and uses bias correction. Final step size per parameter is ratio between first and second order accumulators. Compared to RMSProp, Adam does not have the same high bias at the beginning of training and is seen as a very robust update rule [15].

While the methods used to improve step size vary between these algorithms, empirical experiments show that there is no clear winner among all these update rules [37]. Considering how easy it is to swap between update rules while using modern deep learning frameworks (e.g. Theano [43], TensorFlow [1]), trying different update rules should be done especially if one expects performance gain. Images 5 and 6 in [36] offer a good illustration of how different update rules behave and why we can do better than regular stochastic gradient-descent.

Name	Algorithm
Stochastic gradient descent (SGD)	Update parameters: $\theta \leftarrow \theta - \eta g$
Momentum [33]	Update momentum: $v \leftarrow \alpha v - \eta g$ Update parameters: $\theta \leftarrow \theta + v$
Nesterov-momentum [40]	Update momentum: $v \leftarrow \alpha v - \eta \nabla \mathcal{L}(\boldsymbol{X}, \boldsymbol{y} \mid \theta + \alpha v)$ Update parameters: $\theta \leftarrow \theta + v$
AdaGrad [10]	Accumulate squared gradient: $a \leftarrow a + g \cdot g$ Update parameters: $\theta \leftarrow \theta - \frac{\eta}{\sqrt{\varepsilon + a}} g$
RMSProp [44]	Accumulate decayed gradient: $a \leftarrow \alpha a + (1 - \alpha)(g \cdot g)$ Update parameters: $\theta \leftarrow \theta - \frac{\eta}{\sqrt{\varepsilon + a}} g$
AdaDelta [51]	Counter: $n \leftarrow n + 1$ Calculate update: $\Delta\theta \leftarrow \frac{\sqrt{b}}{\sqrt{a}} g$ Accumulate mean-squared gradient: $a \leftarrow a + \frac{g \cdot g}{n}$ Accumulate mean-squared updates: $b \leftarrow b + \frac{\Delta\theta^2}{n}$ Update parameters: $\theta \leftarrow \theta - \Delta\theta$
Adam [23]	Counter: $n \leftarrow n + 1$ Accumulate 1st order g: $a \leftarrow \alpha_1 a + (1 - \alpha_1) g$ Accumulate 2nd order g: $b \leftarrow \alpha_2 b + (1 - \alpha_2) g \cdot g$ Bias correction: $a_b \leftarrow \frac{a}{1 - \alpha_1^n}$ Bias correction: $b_b \leftarrow \frac{b}{1 - \alpha_2^n}$ Update parameters: $\theta \leftarrow \theta - \eta \frac{a_b}{\varepsilon + \sqrt{b_b}}$

Table 2: *List of common update rules for updating weights in gradient descent per parameter. g is the computed gradient, η is the global learning rate, θ parameters to be updated, ε is a small constant used for numerical stability and α is used for exponential decays. Note that computed gradient can be either mean gradient of whole training batch, a mean of smaller set (mini-batch) or a gradient of a single training sample. Mini-batch training is often used for increased parallelism over using one sample at a time.*

3.4 Overfitting and regularization

One commonly rising issue with machine learning methods is so called *overfitting* [6]. Machine learning methods aim to *generalize* for the given data which means being able to predict labels for unseen data correctly [6], i.e. find the underlying structure and dependencies between variables based on given data. Without need for generalization we could simply store all the samples and labels from the training set and check correct labels when new data arrives, but this very rarely is useful for any practical purpose (e.g. we do not have all possible input features available). Collected data can also include noise, especially if we are dealing with a physical world measurements. Modeling such noise should be avoided as it does not represent the true structure of the data.

Especially with larger models like neural networks the overfitting may be a cause of too high *capacity* of the model. Capacity does not have formal definition but it can be used to describe how well model can model different functions [15]. Low capacity means it can only model functions composed of low amount of functions, while higher capacity allows model to arbitrarily complex functions. However, with too high capacity model might start to approximate noise and other random variations we do not want to model. Hence it is not always optimal to go with the largest model available. See Figure 5 of an example of this behavior in case of a polynomial regression.

This is where previously mentioned evaluation and test dataset come into the picture: We can use the training data (used to optimize parameters) to evaluate the performance of current parameters but this does not represent current generalization of the model on unseen samples, since we are essentially doing recall task. Hence we define a non-overlapping test dataset which is used for the evaluation. The validation dataset (also called *development set*) is not used for actual parameter optimization but for e.g. optimizing hyper-parameters and other non-parameter-optimization methods. Idea is that **test set is not used in training phase in any way** to obtain as realistic performance results as possible. High unbalance in performances between different sets indicate of either overfitting to the training data or bad training/testing data (sets are from different distributions).

3.4.1 Cross validation

Instead of a separate evaluation set a common practice is to use *cross-validation* [6]. In cross-validation, training data is split into different, same-sized chunks one of which is used for evaluation of trained parameters and rest is used for training. One example is *k-fold* cross-validation where data is split into k chunks and training is done k times, each time using different chunk for evaluation and rest for training. Taking average of results of using different evaluation sets should also reduce amount of hyper-parameter overfitting to a single, static dataset. Having only a training and an evaluation set is also a two-fold cross-validation method. Like with validation set, we can detect overfitting from performance with different folds performance varies considerably between folds.

3.4.2 Regularization

There are also approaches to prevent overfitting outside searching for best model and hyper-parameters. A simple heuristic method is to do *early stopping* [6] when using iterative methods. Early stopping stops iterative parameter update loop when minimal error in validation set has been reached or when certain number of steps has been reached. This way we make sure error is minimized for the unforeseen samples. More sophisticated approach is to introduce a *regularization term* [6] into the loss function. There are number of different regularization terms, common of which are L_1 and L_2 norms. For least-squares the regularized loss function [6] is

$$\mathcal{L} = \sum_{i=1}^{N} (\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)^2 + \frac{\alpha}{2} \|\boldsymbol{w}\|^2 \,, \tag{30}$$

which conveniently also has a closed form solution similar to the normal least-squares solution [6]:

$$\boldsymbol{w} = (\boldsymbol{X}^T \boldsymbol{X} + \alpha \boldsymbol{I}_d)^{-1} \boldsymbol{X} \boldsymbol{y} \,, \tag{31}$$

where α is called a *regularization coefficient* which is used to control the strength of the regularization. By introducing regularization term $\|\boldsymbol{w}\|^2$ in the loss function the optimization of said function will balance the minimization between squared errors and this regularization term. In this case the term aims to keep all weight parameters low so that they are more evenly distributed instead of having large spikes, i.e. "all

weight components contribute to the fit and none of them should have too much of total weight". Results of using this regularization can be seen in Figure 5 where regularization caused originally overfitted model to generalize to the data by having similar quadratic shape, which was the underlaying shape of the data".

3.5 Feature crafting and learning representations

There are numerous models to choose from and parameter optimization methods to use, but we still have not discussed about the actual input for the machine learning methods. As defined before, $\boldsymbol{X}$ with N samples is the input that is fed to the model and depending on the task it is used along with $\boldsymbol{y}$ to optimize the parameters. In some cases the inputs are called *feature vectors* or just *features* [6] which are used to represent the observation in a informative but a compact form. In cases where data is representable in a table form with observations on rows and variables on columns, we can feed this directly to almost any of these models. Example of such dataset is Iris-dataset [12] which is an example of classifying flowers based on petal length and other related values. Features include the human intuition about what is good to know for a task and possible domain knowledge on the problem. For example, for audio signal processing computing the Fourier transform could provide good information, or that flower color is good way to distinguish one flower species from another.

However, some data types are not as straight forward to handle, like audio-signal, text documents and images: The number of components per observation might be too large for computations, the number of components varies (e.g. time series) and there may be some crucial dependencies between values. In natural language the context is important (dependency to surrounding words), and in case of images single pixel alone does not tell much (dependency on surrounding pixels). Even then there can be more problems, such as with images where represented object can be at any location of the image. Previous methods would have to learn same object-detection for each location of the image separately.

3.5.1 Manual way: Feature crafting

This is where *feature extraction* happens. We aim to develop algorithms and methods to extract informative feature vectors from original observation using some manually defined procedures. For images this can be detection of corners and computing some

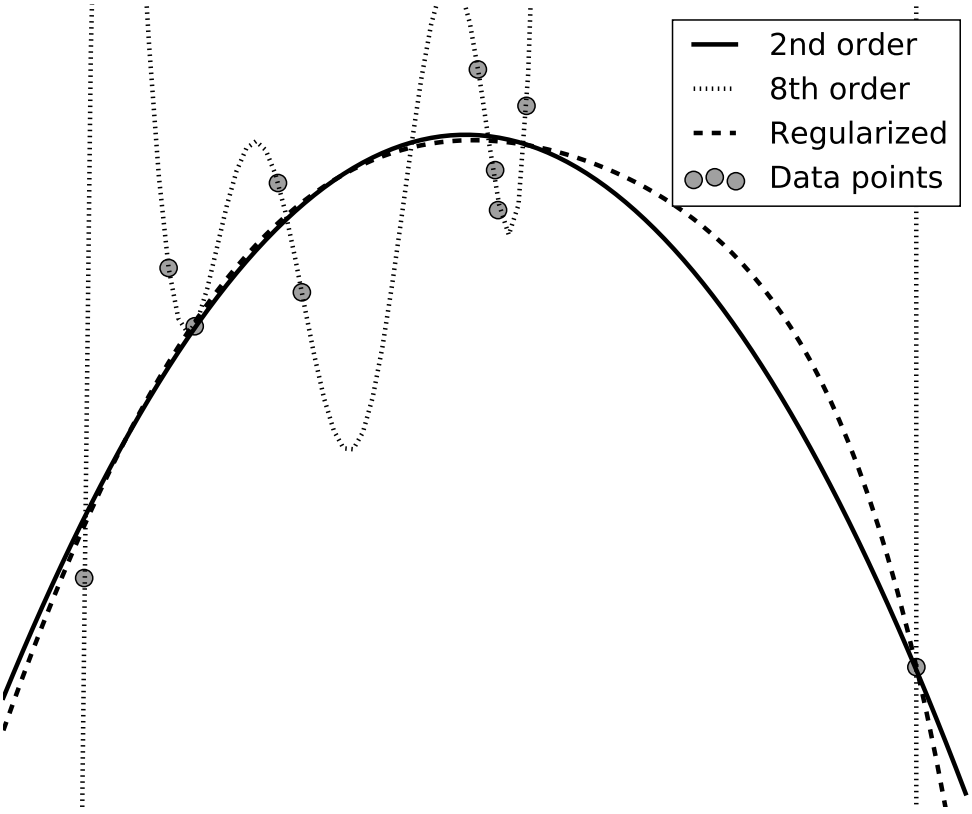


Figure 5: *An example of overfitting. A sample of ten points was taken from a quadratic equation with some uniform noise (gray circles), and two polynomial regressions were fitted. The 8th order model (dotted line) almost perfectly fits to the data but does not have quadratic shape and thus does not generalize well. Meanwhile 2nd order model (solid line) is able to fit into underlying quadratic function. Using regularization (dashed line) with 8th order model and regularization coefficient* $\alpha = 0.0001$ *results to a function similar to the original quadratic equation.*

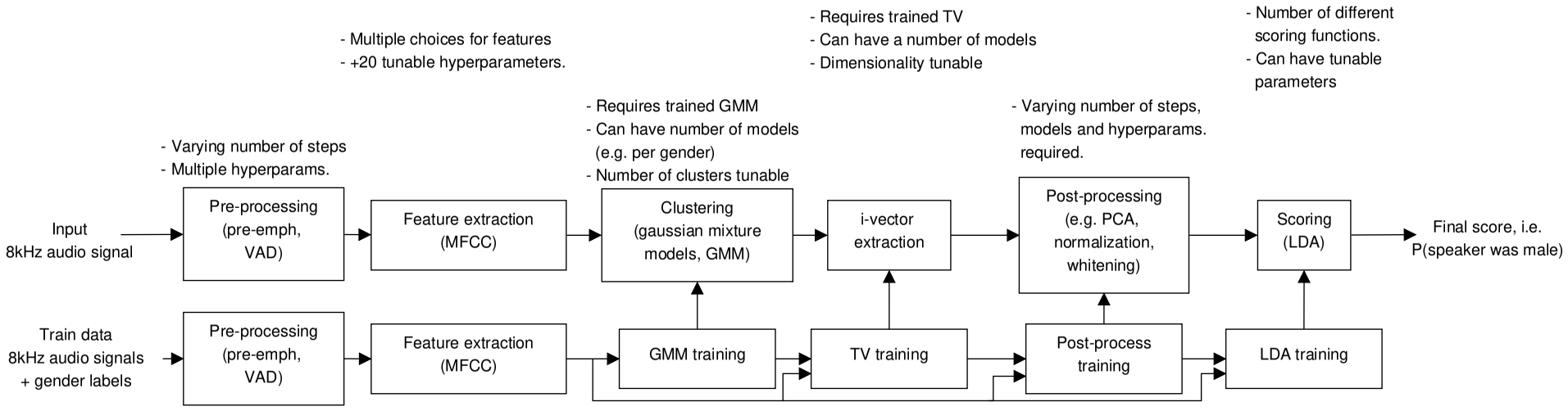


Figure 6: *A gender classifier system described in [21]. Machine learning system can include multiple different steps each with separate set of hyper-parameters that may or may not affect the performance of the system. The feature extraction includes researcher's intuition of what important information we can extract from the raw audio.*

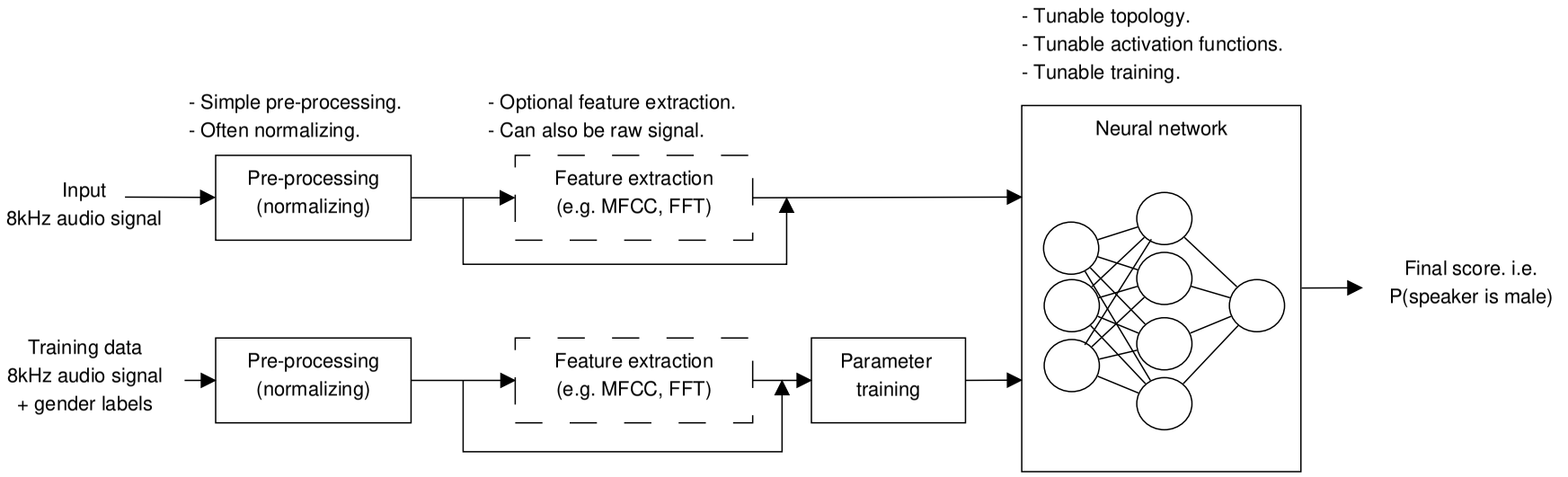


Figure 7: *A gender classifier system tested while running experiments for [21]. Without feature extraction only manually tunable hyper-parameters are in neural network model. Researcher's intuition of how data should be handled is encoded in the topology and functions of the neural network. Instead of extracting features the network learns representations for the raw input audio. Compared to classical machine learning system described in figure 6, there are no dependencies between different parts we need to account for (e.g. how features affect i-vector extraction). However this also leads to hard-to-decipher system, i.e. we do not know how the network solves the problem exactly.*

descriptions for them, and for audio signal a selected amount of frequency components. However, manual feature extraction might discard some information that could be useful for solving the task at hand (e.g. Fourier transform with large windows lacks temporal detail). This is beneficial in terms of computational resources, but arguably we might lose information that could be necessary for the machine learning task at hand.

Using feature extraction in the task requires selection of feature extraction method and possibly comparing multiple different feature extraction methods to find the one providing highest performance. The methods can also include hyper-parameters which can have significant effect on the results. For example in speech processing, one can compute Fourier-transformation of the signal or mel-frequency cepstrum coefficients or any of its variants. One can drop unnecessary frequencies from these results to save space, and/or perform high/low-pass filtering depending on the task. Figures 6 illustrates a task of gender detection from speech using system with feature extraction.

3.5.2 Automatic way: Learning representations

This is where *deep learning* differs from classical machine learning: Instead of crafting the features manually we let the machine learning method learn to extract the meaningful features [15]. Idea is to give raw images and raw audio as an input to the model and let it figure out the proper parameters for handling it. That is, we use the optimization of the parameters to learn the representations for the input data in such a way it benefits solving the task at hand.

Figures 6 and 7 represent a system that has same function, classifying speaker's gender based on their speech, but Figure 6 represents a system using feature extraction and well studied features, clustering, pre-processing and scoring functions. Figure 7 represents a deep-learning variant of this same system. The classical system includes a number of separable steps each of which modify the data in some way beneficial for classifying. The deep learning version includes only few components and it can be reduced to only one major component, the neural network. Systems like these are sometimes referred as *end-to-end* systems, where all processing happens in the major component and no other interference is required (e.g. you give network raw audio signal and it outputs the score you want).

At the first sight the classical system might seem more intuitive and better suited for a

job. After all, it consists of mostly independent parts and research is able to tweak each of them, or even replace them with another blocks to see its effects on results. Indeed this approach can be desired, e.g. in software engineering where separate blocks should only communicate via given channels to minimize complications during development. In machine learning we want to know which algorithms and methods provide best result, and effects of one single part can be hard to confirm when there are multiple different parts with dependencies between each other. For example: *K-means algorithm* [25] (not called k-means in the publication) clusters data-points to one cluster each. There can be data-samples that are on the very edge of being classified to cluster 2 but are classified to cluster 1. We might want to capture this uncertainty and use something like *Gaussian Mixture Models* [6] that produces probabilities of belonging to cluster instead of hard labels, which might improve the performance of the system.

This "era" of deep learning grew popularity from 2007 when Geoffrey Hinton and his student used *Restricted Boltzman Machines* (RBM) to successfully train a large neural network. Similar research has been done in the past as well, e.g. multiple layers of regressions in 1971 [19], but not with such general popularity. The exponential increase of computing resources and available data ("Big Data") are the key parts that made modern deep learning possible [15].

3.5.3 Convolutional neural network

Deep learning systems manifest as a large neural network models with some specific methods and topology that allow neural network to learn handle data at multiple levels of dependency. Instead of modifying separate, distinct parts of the system we modify architecture of the network. One common deep learning method used in image processing tasks is a *convolutional neural network* (CNN) [24]. CNN trains multiple different filters which are used with convolution to process incoming data (see illustration of convolution in Figure 8). These filters are also called "kernels" or "feature maps" [15]. For example, a filter representing a horizontal line will activate (produce higher value) when it is applied on part of image with a horizontal line.

During implementation/usage of CNN layers, the coder specifies number of hyper parameters for the layer, main of which are: Number of kernels, size of the kernels, the stride and the padding style. The output of a CNN layer can be understood as an image with number of kernel channels and height/width depending on the kernel size, stride and padding style. Stride specifies how many steps kernel is moved per one output

(e.g. Figure 8 has stride of 1), and padding style specifies how edges of the images are handled. CNN layers can have very large number of parameters which makes them slower to train, but due to the nature of CNNs they can be run efficiently on GPUs, as they are originally designed for similar image processing.

By applying multiple layers of CNN in a row layers, the later layers will learn higher level representations of the data. For example: the first layer detects simple edges, second layer detects trivial shapes (lines, corners, curves) and further layers combine these trivial shapes into meaningful objects (e.g. a face). Remarkable improvement in image classification and other image related machine learning tasks has been made with networks based on CNN and is a common solution for handling images in neural network models [15].

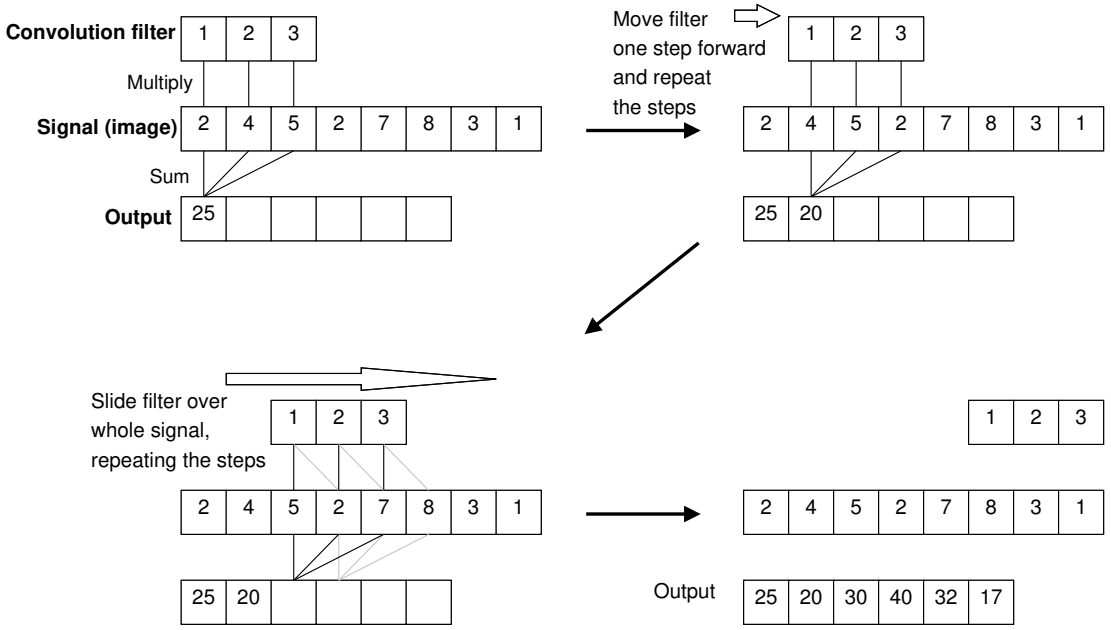


Figure 8: *Illustration of convolution operation in one dimension. This same idea also generalizes to multiple dimensions (e.g. images). In practice one can adjust the number of steps moved (stride) and how convolution behaves at the edges. The illustration shows convolution without padding and only calculates the output for the points where there is an element in signal for each filter element (this is sometimes called "valid" padding). Padding signal from the end with zeros for the length of filter will result to output with same size as input ("same"). Padding signal from start and end will result to output larger than input ("full").*

4 Deep reinforcement learning

Now that we have had a brief look into reinforcement learning, general machine learning and deep learning we can combine them into one. Reinforcement learning algorithms presented in Chapter 2.3 assume and use tabular representation of the state/action values (e.g. a matrix, one element for one state-action pair). This makes designing and analysis of the algorithms easier but such discrete representation is not practical if the number of states/actions is very high. States may also have different representation numerically but underlying information is the same, so it would be useful to have these two states have similar state/action values. e.g. In a game that uses coordinates to define object's position, being in position $(5, 0)$ is likely to be identical to being in position $(5.0001, 0)$,

This can be dealt with by replacing the tabular representation of values with a function approximation, such as a neural network. It has been shown that under certain conditions, linear function approximator converges to an optimal solution under temporal difference learning [45]. Recently a team from Google DeepMind presented a *Deep Q Network* (DQN) [27] which used non-linear approximator, a deep neural network, and successfully learned to play number of Atari video games only based on raw visual input. Since this original publication a great deal of research has been done in area, which some publications call *deep reinforcement learning* (e.g. [27, 26, 22, 39, 29])

4.1 Deep Q Network

Publication presenting deep Q network [27] outlined two problems with replacing tabular state/action value representation with neural networks: Correlations between successive samples and non-stationary network targets/labels. In a video game setting, and in physical world, successive states and frames are very similar and changes between them are minimal which later build up to large changes. This high correlation breaks approximators typically used in pattern recognition, which is to assume data is *independent and identically distributed*. Non-stationary targets are result of agent's policy changing after each update, which can result to significantly different target value next time same input is seen. Normally machine learning algorithms have similar output for the same input, but here the output can be different each time it is fed through the

network and it can constantly change.

4.1.1 Replay memory

First major modification to Q-learning presented in [27] is *replay memory*. Instead of using recorded experiences $e_t = \{s_t, a_t, r_t, s_{t+1}\}$ for training in obtained order, we store experiences to a replay memory $U = \{e_1, e_2, ...e_n\}$ and take uniform samples of experiences for training the network. This breaks the correlations between successive frames and allows higher memory efficiency where same experience is likely to be trained on network a number of times instead of just one. *Prioritized replay memory* [38] improves this replay memory weighting the sampling based on how much was learned from the experience by using the temporal difference error as a weight, yielding higher performance and faster training with DQN in Atari games.

4.1.2 Target network

Second major modification was use of a *target network* (or parameter freezing). Target network is same as the network used to predict the state-action value, but its parameters are only updated at certain intervals. Q-learning requires computing of state-action values $\max_a Q(s_{t+1}, a)$. If we used normal approach, this would cause problems with target changing on each iteration. DQN takes a copy $Q' \leftarrow Q$ of this network every N:th frame ($N = 10\ 000$ in original DQN) and uses Q' to evaluate the value of next state $\max_a Q'(s_{t+1}, a)$. This way the target of network stays mostly same over time and stabilizes the learning. This updating of target network can also be done by annealing parameters of target network Q' towards Q on each step by slightly updating the parameters.

For more minor modifications, DQN used *reward clipping* which clipped the rewards to $[-1, 1]$ to allow DQN to play number of different Atari games without manual reward tweaking, since games can have different orders of magnitude for scores (e.g. number of lives left vs. time survived in seconds). The network consisted of convolution layers which allowed receiving raw image from the Atari's screen, same what human would see, and as a output were all possible actions human player could execute on Atari console. Algorithm 2 includes a rough pseudo code of the DQN system.

As a result, DQN was able to play more than half of the 41 Atari games tested at human-level or above [27] and outperforming best linear learners in all of the games except two. The general conclusion of the publication was that DQN was able to learn

human-like control with minimal amount of per-game manual tweaking. DQN has also received number of improvements like "Dueling Q Network" [48] and "Bootstrapped

DQN" [32] which further increase the performance.

$\theta \leftarrow$ Parameters of the neural network $Q(s_t, a_t)$;
$\theta' \leftarrow$ Parameters of the target neural network $Q(s_t, a_t)$;
$M \leftarrow$ Replay memory. Holds experiences $e_t = (s_t, a_t, r_t, s_{t+1})$;
$l \leftarrow$ Training rate (how many steps per parameter update);
$f \leftarrow$ Target parameter update rate (how many steps per frozen parameter sync);
$\varepsilon \leftarrow$ Probability of selecting random action (ε-greedy policy);
$s_{t-1}, a_{t-1}, r_{t-1} \leftarrow$ Stores previous states, actions and rewards ;
$\gamma \leftarrow$ Discount factor ;
$t \leftarrow 0$;
while *learning* **do**
Get state s_t from the environment (raw image pixels);
Select action $a_t \leftarrow \begin{cases} \text{random action at probability } \varepsilon \\ \arg\max_a Q(s_t, a) \text{ at probability } (1-\varepsilon) \end{cases}$;
Execute action and recieve clipped reward $r_t \leftarrow \max(-1, \min(1, r_t))$;
if s_{t-1}, a_{t-1} *and* r_{t-1} *available* **then**
Store experience to replay memory: M.push$(s_{t-1}, a_{t-1}, r_{t-1}, s_t)$;
end
if *if enough samples in replay memory and* $\mod(t, l) == 0$ **then**
Sample a mini-batch of experiences from the replay memory;
Compute targets for each sampled experience
$y_i = r_i + \gamma \max_a Q(s_{i+1}, a \mid \theta')$;
Update network parameters θ with loss $\mathcal{L} = \mathbb{E}\Big[(y_i - Q(s_i, a_i \mid \theta))^2\Big]$;
end
if $\mod(t, f) = 0$ **then**
Synchronize target network parameters $\theta' \leftarrow \theta$;
end
Update holders $s_{t-1}, a_{t-1}, r_{t-1} \leftarrow s_t, a_t, r_t$;
Proceed to next time step $t \leftarrow t + 1$;
end

Algorithm 2: *Rough pseudocode of the Deep Q Network [27] used to play Atari games. Note that frozen parameters are used for calculating targets instead of current network parameters. Excluding replay memory and frozen parameters, this algorithm is very similar to normal Q-learning. Note that this function does not handle terminal states correctly. Function* $\mod(x, y)$ *is the modulo.*

4.2 Asynchronous deep reinforcement learning

While original DQN performed well in the Atari setup it still required training the network on each game separately. Since the training itself can not be parallelized due to iterative steps, most speedup is gained by using a graphical processing unit (GPU) instead of central processing unit (CPU) to compute extensive matrix computations of a neural network. For example, quick experiments with a dueling DQN [48] agent in ViZDoom [22] show that learning speed is $\approx 20 - 40\times$ faster on a Nvidia GeForce GTX 760 GPU than on a Intel i7-3770k CPU when using Python with Theano [43]. However, the training itself is still happening on one process core, and bigger companies and laboratories can have hundreds of cores and machines available.

4.2.1 Computer-level parallelization

General Reinforcement Learning Architecture (Gorila) [29] allows parallelizing reinforcement learning on multiple computers connected via network connection. Learner nodes have a separate DQN instance running where they compute gradients from their local or global replay memories, parameter server combines gradients from multiple learners into one and learners synchronize their model parameters with the parameter server at fixed interval. This approach further improved DQN performance in most Atari games [29] and was used for training AlphaGo [39] which was able to win world champion in game of Go.

4.2.2 Core-level parallelization

A study by DeepMind [26] focused on similar parallelization but for processor cores instead of multiple computers. The study presents an asynchronous version of previously discussed DQN but more known contribution of this publication was asynchronous *advantage actor-critic* (A3C). A3C uses differences between states' values (increase of state value when moving from state s_t to s_{t+1}) instead of raw state-action values, updating policy by using estimated advantage of each action. A3C uses policy gradient [42] to update the policy. Policy gradient uses the gradient of the loss to update the policy, knowing which way to move parameters to decrease loss, basically using gradient descent (see Chapter 3.2).

The asynchronous part is similar to Gorila, where multiple instances of the same environment and agents were executed. However, only one of the instances performed

parameter updates, while others explored environment, gathered experiences and calculated gradients which were sent to the updater instance. This removes the bottleneck of network connection and allows utilizing computer's cores more efficiently. Figure 9 illustrates this structure and how different parts of A3C communicate with each other, and algorithm 3 outlines the core procedure of A3C learning.

Study of asynchronous deep reinforcement learning methods [26] also include n-step versions of the asynchronous algorithms and a comparison. N-step learning was mentioned in Chapter 2.3 while discussing about temporal difference learning. Instead of bootstrapping to learned value on each time step, we run fixed amount of steps and store experience consisting of observed states, executed actions and received rewards. After running certain amount of steps or reaching a terminal state (e.g. game over) we iterate over the stored experiences starting from the latest experience. This way we know what the reward can be in distant future than instead of just one step ahead and support our learning, since with one-step learning we would have to experience visited states multiple times to move learned information backwards in MDP graph. This also reduces the number of computationally expensive updates on global parameters, where many threads access the same memory. Algorithm 3 illustrates n-step learning by using maximum of t_{max} steps per update, stack a to store the experiences and R to store cumulative reward.

Compared to an improved DQN, A3C outperformed DQN in terms of performance and time it took to train to obtain said performance, even when A3C did not utilize a GPU for speeding up the computation [26]. Asynchronous version of DQN was even able to achieve super-linear performance, performance increased faster than number of threads, which can not be explained only by computational speedups [26]. Study suggests that this is result of parallel, and more efficient, exploration of the environment and different options that reduce the bias of update parameters.

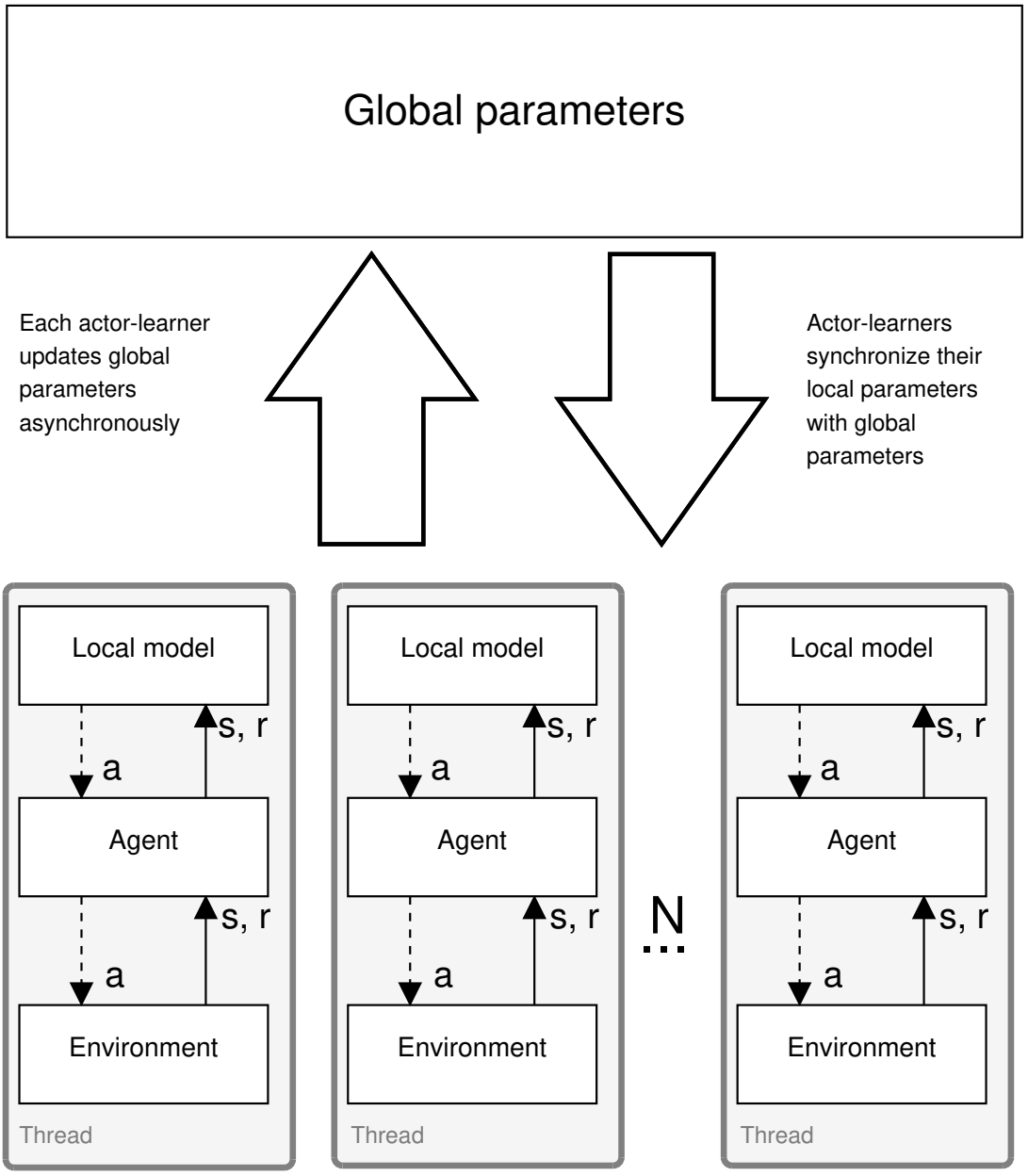


Figure 9: *Graph of an A3C system [26]. Each block runs asynchronously in separate threads. N amount of actor-learner threads interact with separate instances of the environment, accumulate the gradients, update global model parameters asynchronously and finally synchronize local parameters with the global parameters. Compared to single-threaded Q-learning this system learns from more varying experiences (multiple different environments running) which speed up learning super-linearly compared to number of threads used in some cases [26].*

$\theta, \theta_V \leftarrow$ global policy and value estimator parameters;
$\theta', \theta'_V \leftarrow$ local policy and value estimator parameters;
$t_{\text{max}} \leftarrow$ Number of steps included in N-step learning;
while *learning* **do**
Sync parameters $\theta' \leftarrow \theta$ and $\theta'_V \leftarrow \theta_V$;
Initialize stack of experiences $a \leftarrow []$;
Get state s_t from environment;
for t_{max} *steps or until state* s_t *is terminal* **do**
Perform action a_t according to current policy $\pi(a_t \mid s_t, \theta')$ by sampling;
Receive reward r_t from environment;
Push experience (s_t, a_t, r_t) to stack a;
Advance by one step $t \leftarrow t + 1$;
Read next state s_t;
end
Initialize cumulative reward $R \leftarrow 0$;
if s_t *is not terminal* **then**
Bootstrap to learned value of current state $R \leftarrow V(s_t \mid \theta'_V)$;
end
Initialize update gradients for policy $d\theta$ and value $d\theta_V$;
for *length of* a **do**
Pop latest experience $e \leftarrow a$.pop();
Update cumulative reward $R \leftarrow e_r + R\gamma$;
Accumulate gradient for policy with local params.:
$d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(e_a \mid e_s, \theta')(R - V(e_s \mid \theta'_V))$;
Accumulate gradient for value with local params.:
$d\theta_V \leftarrow d\theta_V + \nabla_{\theta'_V}(R - V(e_s \mid \theta'_V))^2$;
end
Update global parameters: $\theta \leftarrow \theta + d\theta$ and $\theta_V \leftarrow \theta_V + d\theta_V$;
end

Algorithm 3: *Pseudo code outlining actor-learner thread of an asynchronous advantage actor-critic system (A3C) [26]. Multiple actor-learners continuously update and fetch same global parameters at fixed interval, but run separate instances of environment where they apply said parameters.*

4.2.3 Optimizing for graphical processing units

While asynchronous methods like A3C and ADQN presented in [26] train reinforcement learning agents faster and with higher performance in terms of reward gained, the publication only uses processor cores for the A3C experiments. While still being faster than DQN trained on a CPU supported by a GPU, A3C leaves the GPU underutilized. Without replay memory like in DQN, each actor-learner thread uses the shared GPU resources in small tasks which not optimal for GPU computations [2].

GA3C [2] modifies the original A3C to utilize performance of GPUs. GA3C only includes one copy of the model(s) used to predict actions instead of multiple copies like in A3C. GA3C consists of three components:

Actors, which are similar to A3C's actor-learners, each of which own a separate environment they interact with. To predict an action, actors submit their states to a prediction queue and wait for a corresponding prediction from a predictor. Actors also submit their experiences to a training queue which are used to update the parameters.

Predictors, which take states from the prediction queue and compute the policy/value predictions using the models on the GPU. Predictors batch multiple samples together before computing the predictions to make use of GPU's capability to handle such parallel tasks well. After computing policy/value predictions, predictor submits these results back to corresponding actor.

Trainers, which take experiences from the training queue, compute gradients and update the models' parameters accordingly. Like predictors, trainers batch multiple experiences together to utilize GPU's parallelization capabilities and to stabilize learning (mini-batch learning).

GA3C can include varying number of each of these components running in parallel on separate CPU cores, which is illustrated in Figure 10 (slightly modified version). Different number of each of these components can result to different performance in terms of predictions per second (PPS), depending on the environment, task, model used and hardware setup [2]. By using high-end GPUs, GA3C can achieve four to 40 times higher PPS than original A3C with same level of performance in the environment [2].

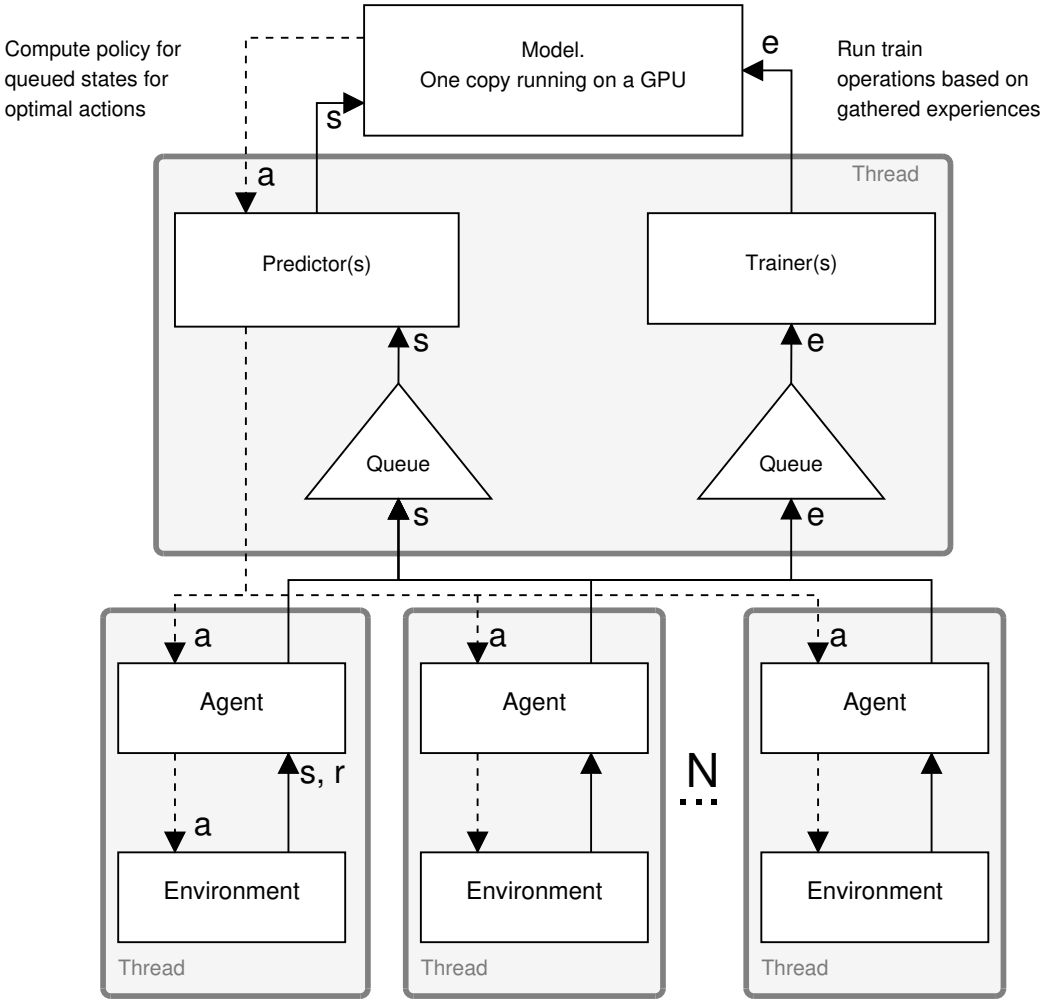


Figure 10: *Graph of a system similar to GA3C system [2] (in original, predictors and trainers were in separate threads). In A3C actor-learner threads use same GPU but each performs separate, small tasks on the GPU which can leave GPU underutilized. In GA3C each actor sends their prediction and training tasks to global queues, which are processed by separate threads interacting with the GPU. These predictor and trainer threads batch multiple tasks together to take full advantage of the GPUs. Depending on the size of the network, GA3C runs 4-40 times faster than A3C in terms of predictions per second [2].*

5 Comparing different input features

With deep reinforcement learning handled we can now approach the research question of this thesis: **Do reinforcement learning agents benefit from using raw image data compared to using other features?** Specifically, does using deep learning to handle raw data such as gray-scale images lead to higher performance compared to high level features like direct object locations? Hypothesis is that using raw information leads to higher performance while using reinforcement learning.

This thesis approaches this question by setting up experiments where different types of input features are compared. Same deep reinforcement learning model is used with different input features, their performance is recorded and then compared against each-other.

5.1 Experimental setup

5.1.1 Environment: VizDoom

VizDoom [22] was used as an environment for the experiments. VizDoom provides convenient tools for an artificial agent to interact with the underlaying video game "Doom" [18]. VizDoom is also very lightweight and can be executed ran in parallel in high numbers to speed up the training. VizDoom provides human-like input and outputs for the agent: The default input (state) for the agent is the frame human player would see while playing the game, and outputs (actions) are set of buttons which to press per each frame.

For further details, VizDoom works in *ticks* or *frames*. By default each tick represents $1/35$ seconds of game time (Doom runs at 35 frames-per-second). Depending on how VizDoom is configured, the game is ran as fast as computationally possible which can reach thousands of ticks-per-second [22]. However with default settings, VizDoom waits for the agent select an action at each step before continuing the game. VizDoom can also be ran in asynchronous mode where the game is constantly ran at given tickrate and VizDoom does not wait for the agent to select actions, which is closer to e.g. human playing the game.

A *step* can be composed of any number of successive ticks, which represents the rate at

which agent receives a state and selects an action. The selected action is then executed till next step. While ticks-per-step can be determined on each action executed, these experiments will use 4 ticks-per-second which represents ≈ 114ms of game time. This value is good balance between computations required to play the game and rate at which agent is provided with information to make efficient decisions. At each step agent can read the state and select the next action to be executed (and how long it should be executed). After executing the action agent receives the reward.

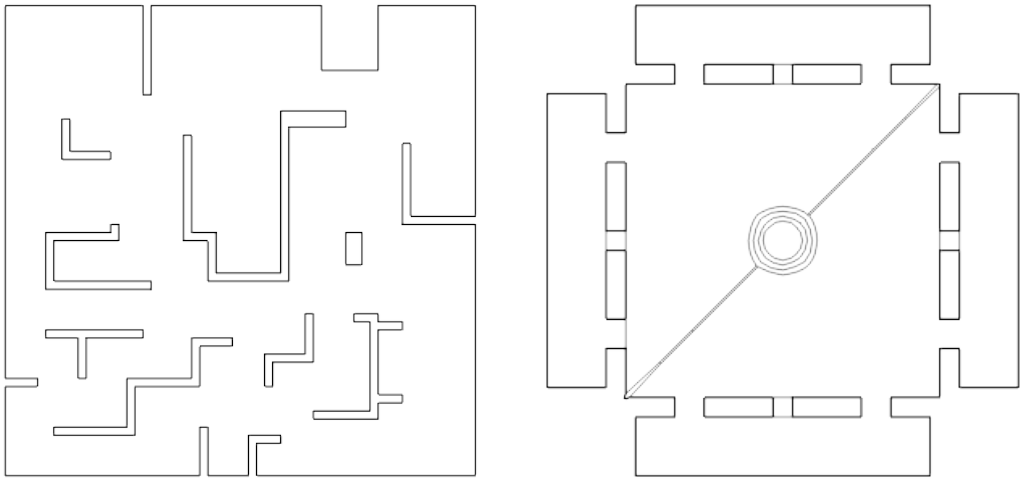
An *episode* is similar to episodes in reinforcement learning, like a single game of chess. A single episode starts from an initial state and ends to a terminal state, e.g. starts from player spawning in the field and ends when player dies. The episode can end to a certain event or when certain amount of ticks have been played. In these experiments we measure performance in terms of episodes. By default VizDoom returns score per episode as the sum of rewards obtained during that episode but we can also define our own score metrics.

VizDoom allows implementing different *scenarios* including different *maps*, which define what is the precise task. It also sets which buttons are available for the agent, e.g. VizDoom comes with some predefined scenarios such as "Health gathering" where agent has to gather health giving pickups ("medikits") and survive till end of the episode, or "defend the center" where agent has to shoot incoming enemy soldiers which try to kill the player. Experiments in this thesis use modified versions of two scenarios provided by VizDoom:

Health gathering supreme, where task is to navigate in a maze and survive as long as possible. Player's health reduces constantly and to reach maximum score player has to collect medkits that give more health, but also avoid "poison" pickups which hurt player. The original reward scheme gave agent reward for each tick player was alive and lots of minus reward when player died. This proved to be too difficult for agent system to learn, so reward scheme was modified to give reward based on how health changed: If amount of health increased the agent received a positive reward and vice versa. Score per episode is defined as ticks_survived + health_left. This way agents can be compared against each other even if they do not survive an episode and when they survive an episode. Allowed buttons consisted of moving forward, turning left, turning right and all of their combinations. Map layout is shown in Figure 11.

Deathmatch, where player is spawned in an arena along with a number of Doom

enemies (same player would meet in the original game). Goal here is to kill as many enemy soldiers as possible, each kill being a positive reward. The player starts with a pistol but there are also stronger weapons available at map's sides, as well as medkits and armor pickups. The layout of the map stays the same so the agent can learn to navigate to certain area of the map to get a strong weapon, but the starting position of the player and the enemies are randomly chosen. Score per episode for this scenario is the number of enemies killed. Allowed buttons consisted of moving forward, turning left, turning right, firing the weapon and all of their combinations. Map layout is shown in Figure 11.



(a) Health gathering, map layout

(b) Deathmatch, map layout

Figure 11: *Top-down view of the scenarios' maps. Health gathering supreme scenario has a static maze player takes constant damage, and thus has to collect randomly spawning medkits to survive longer. Deathmatch scenario has a large arena where enemies constantly spawn, and smaller rooms at each side with different pickups (weapons, medkits, armor and so on).*

5.1.2 Agent: Asynchronous advantage actor-critic for GPUs

The agent of the experiment uses GPU asynchronous advantage actor-critic (see Chapter 4.2) for learning and selecting optimal actions. The implementation is based on open-source implementation of GA3C [1] provided by the authors [2] which was then modified to run VizDoom instead of OpenAI Gym. Dynamic balancer for optimal performance was disabled and each experiment is ran with 32 instances of VizDoom running parallel with three trainer threads and a single predictor thread.

[1] `https://github.com/NVlabs/GA3C`

For approximating policy and value functions the agent uses neural network with following architecture, starting from the layer which receives the input:

1. Convolutional layer. Kernel size 8, 16 kernels, stride of 4, ReLU activation.
2. Convolutional layer. Kernel size 4, 32 kernels, stride of 2, ReLU activation.
3. Fully connected layer, 1024 nodes, ReLU activation.
4. Layer for outputs. One linear output for the value function and one softmax output for each action.

All inputs and rewards were normalized to roughly $[0, 1]$ (see following chapter). This network structure is similar to the one described in [26] where A3C was able to navigate in a 3D maze, except number of nodes of the fully connected layer has been increased to 1024 from 256. The original publication also included A3C using LSTM layers, but this was not implemented in the current state of the GA3C. Since GA3C network receives states from multiple environments in an uncertain order, each state would have to be accompanied with the hidden state obtained from last state, which can be really ineffective without an appropriate implementation.

Network parameters were trained using Adam [23] (see Chapter 3.2) with learning rate $\eta = 7 \cdot 10^{-4}$, $\beta_1 = 0.95$, $\beta_2 = 0.999$ and $\varepsilon = 10^{-6}$. For entropy regularization term $\beta_{\text{reg}} = 10^{-2}$ was used. Both learning rate and entropy term were linearly decreased over training with lowest value at $\eta = 10^{-5}$ and βreg $= 10^{-4}$ at the end of training ($1M$ training steps).

Experiments were ran on a single server computer with Intel i7-3770k CPU, Nvidia GTX 760 GPU, Ubuntu LTS 16.04, CUDA 8.0 (with CuDNN 5), Python 3.5, TensorFlow 1.0 and VizDoom 1.1.0.

5.1.3 Input features to compare

Four different input feature schemes were implemented to compare the performance between raw and more refined input features. VizDoom provides means to implement all of these efficiently. For example pictures see Figure 12.

Gray image, which represents the most raw form of visual input. This is the image human player would see while normally playing the game, except converted into a

gray-scale image and resized to size of 64×48. This image size provides enough information for agents to function in the game, albeit items at longer distance will be hard to detect.

Depth map, which is also a gray-scale image with size of 64×48. However in this case each pixel represents the distance to the object drawn in the pixel. This input feature is slightly more refined compared to the gray image as it is not what humans would see, but a robot with depth sensors could receive similar input.

Label map, which is a 64×48 image with two channels. VizDoom provides a buffer which assigns each visible object with an ID and a name, and this is further refined by assigning these objects in one of two channels depending on the scenario. In health gathering supreme, medkits are assigned in channel 1 and poison pickups are assigned to channel 2. In deathmatch scenario the enemies are assigned in channel 1 and all pickups are assigned to channel 2. VizDoom does not provide labeling for walls and ground. This input type is similar to pixel-level labeling done for some machine learning tasks, e.g. dataset for self-driving cars [35]. In practice providing something like this would have an error rate, and thus the flawless information provided here can be considered oracle information (something we are not supposed to have).

Direct features, which differ from other input features by not being an image. Instead this is a combination of horizontal depth scan and location of objects. Horizontal scan line is the midmost horizontal line in the depth image, and location of objects is fed by giving the distance to the object and angle versus player's aim direction. This is the most feature crafted way of giving state to the agent out of all tested input features. An input feature like this could be a result of using computer vision to detect different objects and estimating their location and distance, plus using a sensor that scans the distance in one horizontal line. In this experiment this info would be flawless, and hence could be considered an oracle information.

The experiments also include combinations of two of the first three input features (i.e. gray+depth, gray+label and depth+label). Figure 12 shows example images of the three first input features in same states in both scenarios.

All values were normalized to roughly $[0, 1]$ or $[-1, 1]$ range. All values in gray images and depth maps were divided by 255 as they were represented by unsigned 8-bit integer. Label maps did not require normalizing as values were in $\{0, 1\}$. For direct features

the distance from object to player was limited to 1200 game units and then normalized by dividing with 2. Angle between player aim direction and objects was in interval $[-1, 1]$. For health gathering supreme scenario, reward was $\Delta\text{health}/20$ (max health is 100, player heals roughly $10 - 20$ health per medkit). In deathmatch scenario no reward normalizing is required, as one kill equals 1 reward.

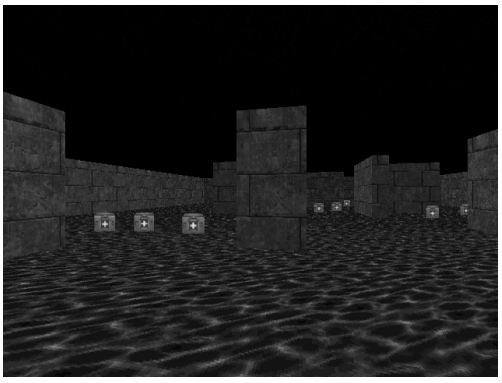
Gray, depth and label input features are all processed with network described earlier by using convolutional layers. Direct features are separated into two inputs: Horizontal scanline and list of objects. Horizontal scanline is fed through a fully connected layer. Objects array is constructed of max 30 valid visible objects (e.g. pickups, enemies). Each object is represented by a one-hot encoded vector representing object's type, the distance to the player and the angle between player's view direction and object-player vector. These arrays were combined into an $30 \times D$ matrix, which was then processed row-by-row with an convolutional layer with 16 filters. The whole network consisted of the input layer and one fully connected layer of 1024 units, essentially only replacing the convolutional layers of the network used with other direct features.

5.2 Results

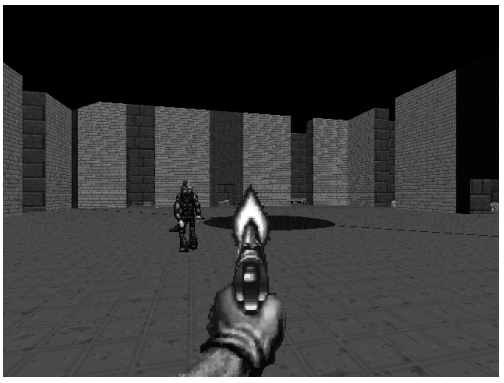
For each scenario and input feature the experiment was ran five times due to variance between runs. Each experiment lasted for 1M parameter update operations, and performance of the agent was evaluated every $20\ 000$th parameter update. An evaluation consisted of 200 episodes without training operations, and final score per evaluation was the average of the score metric of the scenario over these episodes. This average will be referred as "evaluation score". During evaluation the action was sampled from the action distribution like during training. We also experimented by using the action with highest probability, but the end results did not change.

In total there is 48 evaluations per training run instead of logical $1\ 000\ 000/20\ 000 = 50$ evaluations: The initial state (training step 0) is not evaluated, and last evaluation did not always occur due to asynchronous nature of the GA3C (server quit before evaluation was done). This was noticed late in experiment regimen, and there was no time to redo experiments.

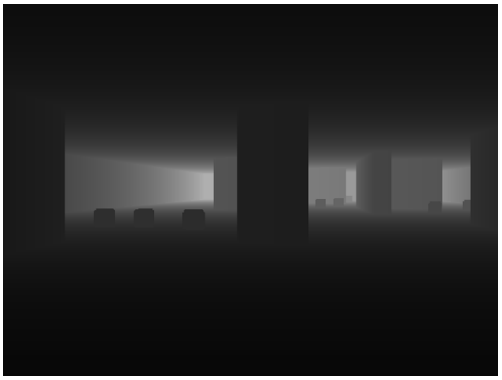
Figure 13 shows medians of the evaluation scores versus number of training steps done. Standalone input features and combinations are plotted in separate plots.



(a) Health gathering, gray-scale image

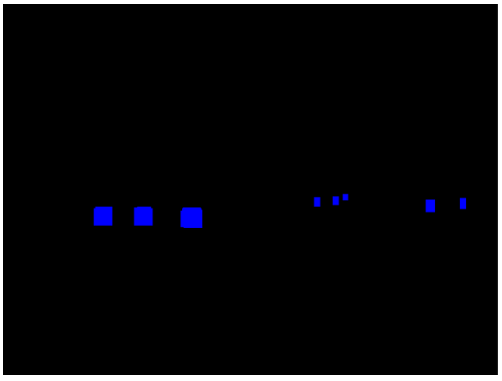


(b) Deathmatch, gray-scale image

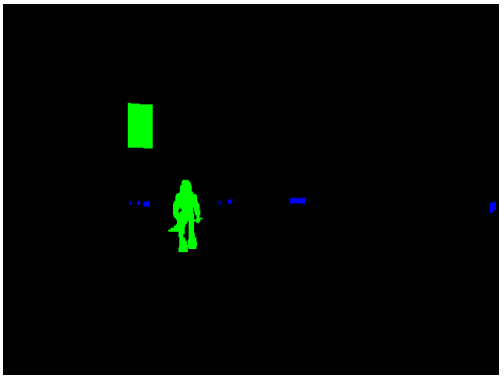


(c) Health gathering, depth map

(d) Deathmatch, depth map

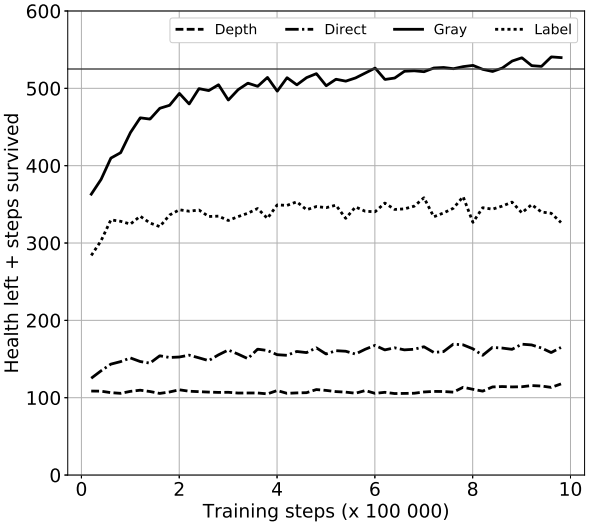


(e) Health gathering, label map

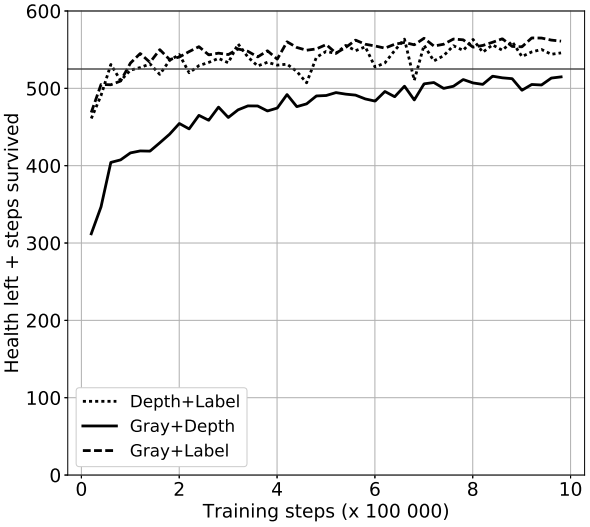


(f) Deathmatch, label map

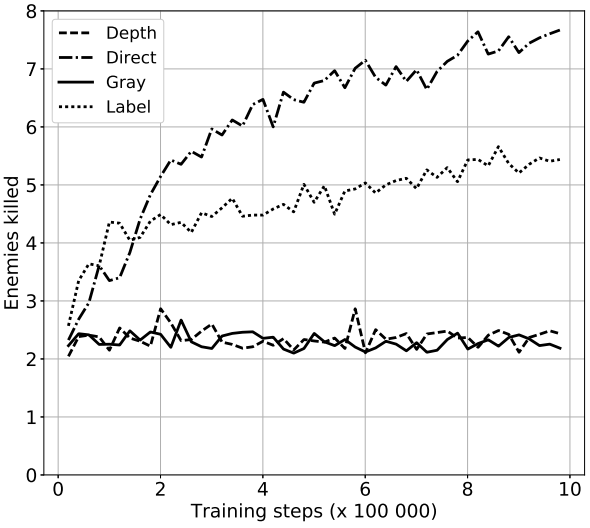
Figure 12: *Images of the two different scenario with three input-feature types. HUD elements (e.g. hand and gun) are not visible to the agent.*



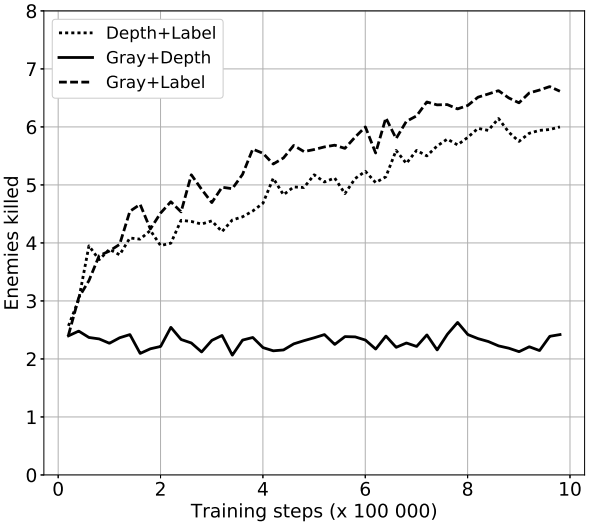
(a) Health gathering supreme, individual inputs



(b) Health gathering supreme, combined inputs



(c) Deathmatch, individual inputs



(d) Deatchmatch, combined inputs

Figure 13: *Experiment results in both scenarios. Each line represent the median of the five runs done per input feature. The variance of score between runs is discussed in the text. Highlighted horizontal line at score 525 in "health gathering supreme" scenario indicates the point where agent survived an episode.*

In "health gathering supreme" scenario gray-scale image stays above other options during whole training, managing to keep agent alive after $\approx 70\%$ of the training (score 525). Label map achieves around half of the score of gray-scale image (≈ 350), and depth and direct features are not able to collect many medikits during episode (below 200). Using combinations of input-features, gray+label and depth+label inputs both quickly survive the episode and also keep above 25 health-points (score ≈ 550). Gray+depth input does not manage to survive episodes, and stays below the performance of using gray image alone. Most experiments had low variance, except depth map and gray+depth (See Appendix B). Gray map could reach higher score during experiments, and gray+depth often has median score but can also have very low score.

In "deathmatch" scenario direct features and label map achieved highest score constantly with small variance (virtually same results on each run), with gray and depth images not improving over time. Depth map reached score closer to label map but only in one of the five runs. In combined inputs, depth+label and gray+label were able to obtain score of 7 at the end of training, while gray+label did not improve over time. All input features have small variance, except for depth map (see Appendix B).

5.2.1 Discussion of the results

Based on these results, using gray image alone is not the most robust or highest performing input feature which is against the original hypothesis. In "health gathering supreme" scenario, gray-scale image performed above all other input features across all training runs. However in "deathmatch" scenario, higher level features such as the pixel-labeled image and direct features provide highest performance robustly during different training runs, while agents with low level features do not manage to improve over time.

Low performance of higher level features in health gathering supreme could be partly explained by feature crafting: Created features did not include some information that was required for agent to learn optimal behavior. It should also be noted that these high level features are perfect and akin to oracle information (something we should not be able to have): Perfect pixel labeling would be very difficult to do without provided information, and extraction of object locations in image could be even harder. Nevertheless, these results show that agents are able to achieve higher performance if the provided information is noiseless.

These results align with existing studies. SLAM-augmented deep reinforcement learning [5] reported similar results, where higher level features did improve the performance of the agent but these higher level features had to be flawless and thus oracle information. By adding noise to the oracle information the performance dropped to the baseline level of using gray-scale image with DQN and dueling-DQN.

Outside objective results the convenience of using low level features can also be accounted for. During these experiments, implementing model for gray and depth map was straightforward. Label map required additional processing of information, and direct features required completely new network model. Input features also required different amount of computational resources to train. Experiments with gray or depth map took around 5 hours, labeled took $6-8$ hours and direct features took $7-9$ hours (due to size of the input). Other features than gray and depth maps might also be hard to obtain in other than simulated environments. Gray image can be obtained with a regular RGB camera, and depth map can be obtained with multiple cameras and/or laser sensors. Label map requires additional processing of assigning pixels in different classes, and direct features would require robust detection of objects in the scene as well depth sensor. However these results are highly dependent on implementation, software and hardware.

6 Conclusion

This work presents introduction to deep reinforcement learning and required fields, namely machine and deep learning. Specifically this thesis focused on using refined, high level input features versus using raw information from sensors while training deep reinforcement learning agents. In the past high level features were used for their small size, but in present day methods exist for efficient use of raw information such as RGB and gray-scale images (deep learning). In this thesis these two types of input features were compared in a classical video game "Doom".

During experiments two different scenarios/tasks were used, and four different input features were defined ranging from low level gray image to high level direct features (object locations). The experiments show that there is no single input feature that would allow high, constant performance in all situations. Using gray-scale image in one scenario led to stable, high scoring results while using pixel-labeling and object locations resulted to high scores in another scenario.

Higher level features used here included included oracle-level information as they were flawless object locations in 3D space and their labels. Based on this, results and a related study [5], higher level features provide higher performance while training deep reinforcement learning agents, but this information has to be very accurate which can be difficult to obtain. There are also cases where the higher level features might not reach high performance compared to low level features. Lower level features are also more convenient to use, as they do not require additional designing and be obtained easily (RGB-camera, laser for depth, et cetera).

To further confirm the results of this work additional experiments could be run with different setups: The network size was relatively small in these experiments, and large network could have higher capacity for processing raw input like gray image. Other type of reinforcement learning methods could be tried other than A3C. Additional scenarios and environments could be tried, like DeepMind Lab, especially something with more visual complexity such as lighting. Additional input features should also be tried, as only one non-image type of input was tried. For future work, adding random noise to the high level features could be tried to see how robust these reinforcement learning methods are against noisy inputs.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: `http://tensorflow.org/`.

[2] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Reinforcement learning thorugh asynchronous advantage actor-critic on a gpu. In *International Conference on Learning Representations*, 2017.

[3] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.

[4] Richard Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957.

[5] Shehroze Bhatti, Alban Desmaison, Ondrej Miksik, Nantas Nardelli, N Siddharth, and Philip HS Torr. Playing doom with slam-augmented deep reinforcement learning. *arXiv preprint arXiv:1612.00380*, 2016.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[8] Valve Corporation. Dota 2. URL: `http://blog.dota2.com/`.

[9] Geoffrey E. Hinton David E. Rumelhart and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[11] Blizzard Entertainment. Starcraft 2. URL: `http://us.battle.net/sc2/en/`.

[12] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[13] David E Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley Publishing Company, Inc., 1989.

[14] Faustino J Gomez and Risto Miikkulainen. Solving non-markovian control tasks with neuroevolution. In *IJCAI*, volume 99, pages 1356–1361, 1999.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016. URL: `http://www.deeplearningbook.org`.

[16] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649. IEEE, 2013.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[18] id Software. Doom (1993).

[19] Alexey Grigorevich Ivakhnenko. Polynomial theory of complex systems. *IEEE transactions on Systems, Man, and Cybernetics*, 1(4):364–378, 1971.

[20] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *International joint conference on artificial intelligence (IJCAI)*, page 4246, 2016.

[21] Anssi Kanervisto, Ville Vestman, Md Sahidullah, Ville Hautamäki, and Tomi Kinnunen. Effects of gender information in text-independent and text-dependent speaker verification. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, 2017.

[22] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *Conference on Computational Intelligence and Games*, pages 1–8, Sept 2016. doi:10.1109/CIG.2016.7860433.

[23] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[24] Yann LeCun et al. Generalization and network design strategies. *Connectionism in perspective*, pages 143–155, 1989.

[25] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[26] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *International Conference on Machine Learning*, 2016.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[28] George E Monahan. State of the art—a survey of partially observable markov decision processes: theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[29] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. In *International Conference on Machine Learning, Deep Learning Workshop*, 2015.

[30] Rockstar North. Grand theft auto v. URL: http://www.rockstargames.com/V/.

[31] OpenAI. Openai universe. URL: https://universe.openai.com/.

[32] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. *Neural Information Processing Systems*, 2016.

[33] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[34] Craig Quiter. Deepdrive. Accessed: 5.2.2017. URL: https://www.youtube.com/watch?v=1uURlRKfLqY.

[35] Stephan R. Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *European Conference on Computer Vision (ECCV)*, volume 9906 of *LNCS*, pages 102–118. Springer International Publishing, 2016.

[36] Sebastian Ruder. An overview of gradient descent optimization algorithms. Accessed: 29.11.2016. URL: http://sebastianruder.com/optimizing-gradient-descent.

[37] Tom Schaul, Ioannis Antonoglou, and David Silver. Unit tests for stochastic optimization. *International Conference on Learning Representations*, 2014.

[38] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations*, 2016.

[39] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[40] Ilya Sutskever, James Martens, George E Dahl, and Geoffrey E Hinton. On the importance of initialization and momentum in deep learning. *International Conference on Machine Learning (3)*, 28:1139–1147, 2013.

[41] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[42] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *Neural Information Processing Systems*, volume 99, pages 1057–1063, 1999.

[43] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL: `http://arxiv.org/abs/1605.02688`.

[44] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 2012.

[45] John N Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE transactions on automatic control*, 42(5):674–690, 1997.

[46] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[47] Oriol Vinyals and Quoc Le. A neural conversational model. *International Conference on Machine Learning Workshop*, 2015.

[48] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *International Conference on Machine Learning*, 2016.

[49] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge England, 1989.

[50] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C Courville, Ruslan Salakhutdinov, Richard S Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, volume 14, pages 77–81, 2015.

[51] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

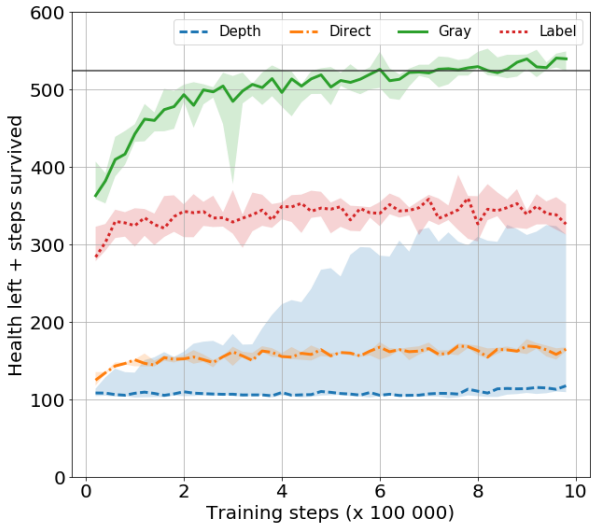
Appendices

A Hyperparameters used in the experiments

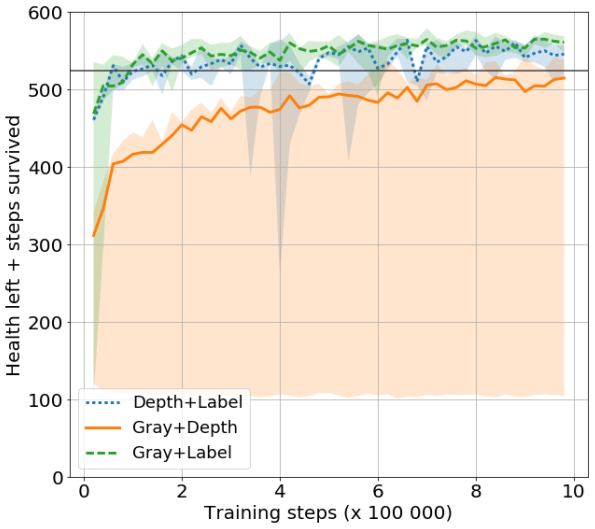
Parameter	Value	Description
	Experiments	
Experiment repeats	5	Number of repeats per scenario $\times$ input.
Experiment length	1 000 000	Number of parameter updates per experiment.
Evaluation rate	20 000	Number of parameter updates between evaluations.
Evaluation length	200	Number of episodes to play per evaluation.
	VizDoom	
Frames per action	4	Number of in-game frames (ticks) per action.
	GA3C (model)	
Discount factor γ	0.99	Discount factor of rewards (see Chatpter 2.3)
Input image size	64×48	Size of the images given to the GA3C agent.
Mini-batch size	32	Number of experiences used per parameter update operation (see Chapter 3.3).
n-step lookahead	5	Number of steps per experience (see Chapter 2.3)
Reward clipping	$[-1, 1]$	Interval for reward, clipped if outside this.
Number of components	$32/1/3$	Number of agents, predictors and trainers (in same order).
β_{reg}	$[10^{-4}, 10^{-2}]$	Interval of entropy regularization term, annealed linearly towards lower end.
	Adam (network optimizer)	
Learning rate η	$[10^{-5}, 7 \cdot 10^{-4}]$	Interval of global learning rate, annealed linearly towards lower end.
β_1	0.95	β_1 (α_1 in Table 2) parameter of the Adam optimizer.
β_2	0.999	β_2 (α_2 in Table 2) parameter of the Adam optimizer.
ε	10^{-6}	Small constant for stabilizing division in Adam.

Different hyperparameters used during the experiments as well values related to experiment setup.

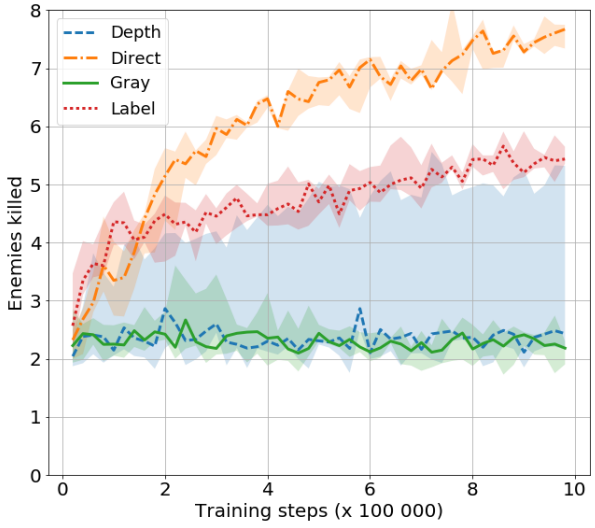
B Variance in experiment results



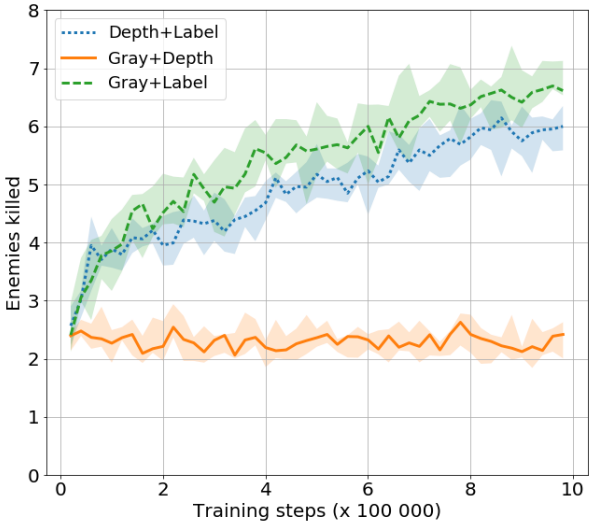
(a) Health gathering supreme, individual inputs



(b) Health gathering supreme, combined inputs



(c) Deathmatch, individual inputs



(d) Deatchmatch, combined inputs

Experiment results and their variance in color. Line represents the median of the results, and opaque area represents interval between minimum and maximum average score over all experiments.