**BEREITS EINGEBRACHT**

* Valentins & Melisas Präsi
* DQN Paper 1 & 2

**NOCH EINBRINGEN**

* Heidemanns Slides
* Batchnorm Paper
* DDPG paper
* Prioritized Experience replay
* Count-based exploration
* Double-Q
* AI a modern approach
* Meinen und Leons grant-text
* Lange auf Universe eingehen!!
* <http://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html#References> das hier und alle seine references!!! Der geht auch länger auf den unterschie dzwischen off-policy und on-policy, modelfree, all den kram! undso ein!! [bei model-free allerdings bspw theoretischer sein, dass model-based den mdp re-createn will]

[github commit signen/verifizieren]

[youtube links zu aktuellen runs]

Introduction

This thesis is about …

Also relevance! What is there already, …

The goal of this work was twofold: first, to convert the game at hand into a reinforcement learning problem, and second to learn on exactly that.

**Neural networks [später]**

A basic understanding of neural networks is presupposed for the reader of this thesis. Especially relevant are CNNs and RNNs, as they are both used here. *(In diesem und dem folgenden Abschnitt wird grundlegendes Wissen über Softwarearchitektur und -entwicklung sowie über Webtechnologien vorausgesetzt. Es werden primär Besonderheiten der verwendeten Paradigmen und Technologien und insbesondere deren Unterscheidungsmerkmale zu oft vorherrschenden Alternativen diskutiert, quote alex’ thesis, da war es aber in der Zusammenfassung am Anfang)*

**Q-learning**

[<https://github.com/ahoereth/ddpg/blob/master/docs/report/report.md>]

[dass both dqn and ddpg off-policy sind, daher supervised pre-training zu nem gewissem grad was bringt, even though es härtestens exploration einschränkt]

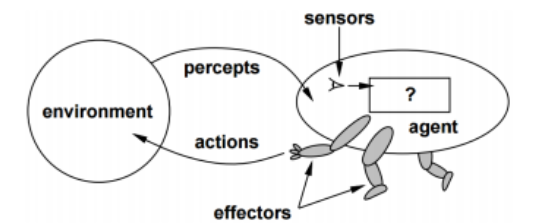
[dass universe nen mdp sehr genau spezifiziert]

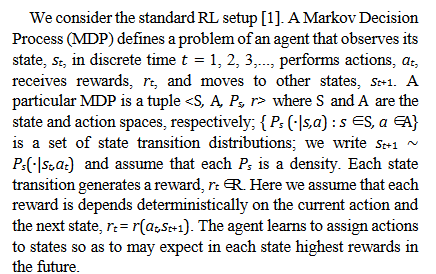
[am anfang GAR nicht auf neural networks eingehen, nur über universal function approximators reden, auch über gradients, und DANN DQN (replay buffer to break temporal correlations), und dann die ganzen anderen changes?]

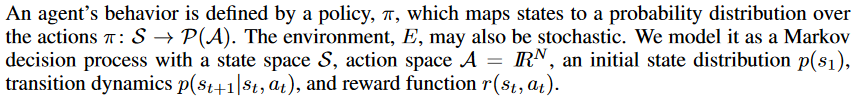
Machine Learning can mainly be subdivided into three main categories: Supervised Learning, Unsupervised Learning, and Semi-supervised learning. The first deals with direct classification or regression using labelled data (i.e. it uses pairs of datapoints with their corresponding category or value). In unsupervised learning, no such label exists, and the data must be clustered into meaningful parts without any knowledge, by for example grouping objects by similarity of their properties.

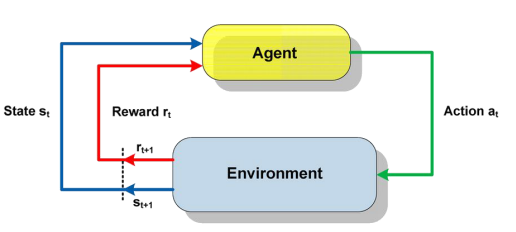
What will be mainly considered in this thesis will be a certain kind of semi-supervised fflearning: Reinforcement learning. In Reinforcement Learning (RL), instead of labels for the data, there is a *weak teacher,* which provides feedback to the actions the agent took

In semi-supervised learning, there are either only sparse labels (labels for only a subgroup of the data), or something else: a *weak teacher*. One kind of semi-supervised learning is Reinforcement Learning.

The metaphor behind Reinforcement Learning that of an agent, interacting with an enviroment (discretized into states). The agent performs actions, changing the state of the environment [output], and receiving rewards. (he percepts the envronment, input) [s,a,r,s’ und das bild rechts einbringen] (direkt(??) die metapher zum auto ziehen, dass da die straße die envriomnet ist und der agent der fharer des autos)

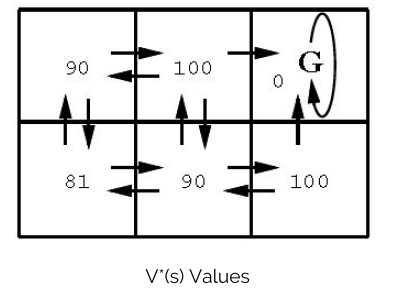
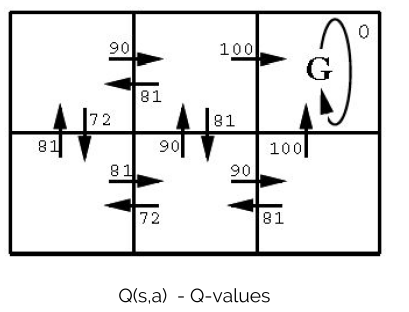




* Weakly inspired by behavioral psychology and the Dopamine system in humans
* Goal: find optimal actions given perceptions
* Markov decision process! <https://en.wikipedia.org/wiki/Markov_decision_process#Definition>
* If an agent knew all possible rewards&transitions from all states, it could perform offline learning using this MDP. (s&a -> s’&r)
* Problem is, transitionfunction & reward are unknown, so an agent needs to learn that → try it out → RL (that’s why its semi-supervised)
* [offline is in feasible in racing games since some states are not visited by chance]

RL can again be subdivided into model-based and model-free. In model-based RL, one tries to approximate the underlying model/MDP by figuring out the transitions & rewards, reconstructing the two unknown functions. Model-free RL doesn’t care about the actual transition function, but only reward&value of state are important.

→ VORHER schon V(s) und Q(s,a) erklärt haben

* [on-policy vs off-policy; Q-learning vs SARSA, Q-function/actionvalue function vs state-value-function, actorcritic vs nonactorcritic]
* What we do is approximate the Bellmann-Equation: (muss fallen) we compare what we thought would happen with what actually happened
* Active vs passive: in passive RL, the agent acts upon an unkown policy that it doesn’t choose himself, learns the value of a STATE (computes all V(S)’s = tootal amount of expected accumulated reward, starting in state S)... however we don’t know how to get there without knowing the policy
* Temporal difference learning approximates V(s) = r(s,p(a)) + yV(s’). With each new sample, we update our V(s) (however we want to do action selection, which we can’t in this case!)
* → for action selection, we want to consider the Q-values instead of the Values → Q(s,a)
* Start with Q0(s,a) = 0, then we update the Q(s,a) with every state/action-pair (BELLMAN-equation) 
* Q(s,a) = alpha \* R(s,a) + y\*maxQ(s’,a’)
* We update only the actions we’re taking! [darauf zurückkommen bei der Q-table und wie das beim DEEP Q-learning ist]
* [dass wir MAXen, ist das nicht bei double-Q gesolved?]
* Now we have the problem of action selection expolration vs exploitation, epsilon bei epsilon-greedy [bzw im countbased paper anders, ist das etwa “epsilon = Q + 1/numvisits”?
* HEIDIS SLIDES !!!!!!!

Policy gradient algorithms utilize a form of policy iteration: they evaluate the policy, and then follow the policy gradient to maximize performance

[source hiervon maybe <http://ai.berkeley.edu/lecture_videos.html>]

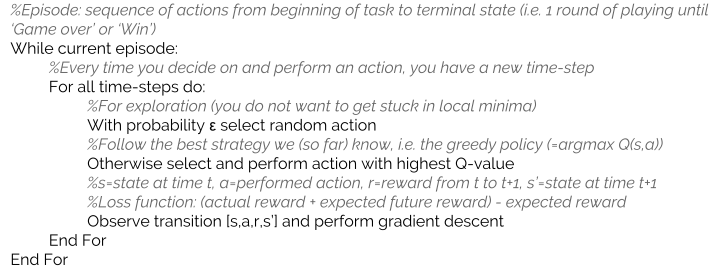
[unterschied SARSA & Q-learn: <https://github.com/ahoereth/ddpg/blob/master/exploration/FrozenLake.ipynb>] (übrigens, werden bei SARSA die werte immer höher? Weil dann wäre es keine bellmann? Oder ist der auch negativ?) In Q-learning, Future Q values are chosen according to the current policy. Q-Learning is greedy: Future Q values are chosen purely by their value.

[Credit assignment problem erwähnen]

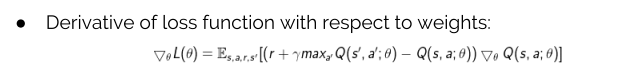
Q-learning in neural nets

Principle here: Our Q-function returns a value, given state and action. This value is periodically updated using the Bellman-equation. In standard RL, one can imagine this Q-function as a table. However in Q-learning in Neural Nets, this Q-function (, evaluating state and action, yielding a Q-value) is approximated using a neural network.

We want to guess the underlying, real Q\*(s,a) = max\_pi E[r1+yr2+yyr3+... | pi], which is the value of an action in a state given the optimal policy. We approximate that [bessere darstellung, woher kam die nochmal] [auch den Unterschied zwischen state of actor und state of environment]

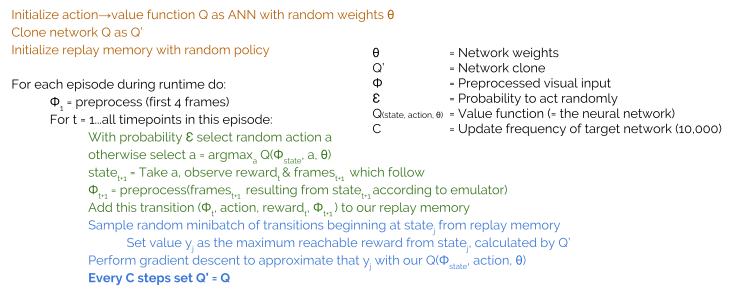
Sooo the algorithm is 

* Without random actions the agent would be strongly biased on performing the first action ever yielding a positive reward and get stuck in a local minimum very quick… → However there are smarter choices!! → intrinsic motivation paper



* ← Dabei vielmehr auf Bellman Equation eingehen!
* The error is the difference between what we computed and what we got +statevalue, and this is then update (but only this!!)

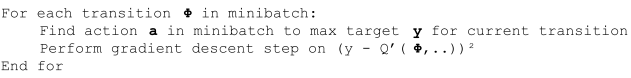
However nobody ever achieved any good results with this, until [minh et al] came with their replay memory & target network



Sooo yeah, important stuff here are replay memory & target network. Also, its more of a markov assumption here: Atari games are perceptually aliased, so its impossible to infer the entire game state using only one screen. Thats why they also have the history. Agent’s state != environment’s state. Doesn’t work too well for some games. Soo the input to the ANN here is a sequence of states. Also, in contrast to previous approaches, all 18 Q-values are calulated (in the inference) in the same network step.

We have the replay memory to “break correlation” (learning from consecutive samples steers network into a direction (“steer right, steer right, steer right, … hey, just generally steer right!”) → smoothes out oscillation

Clear separation of application and learning!  
In the learning step, we assume that our max\_a’ Q(s’,a’) is a good estimate for the value of the folgestate.. So it gets better and better, we iteratively approximate the real, underlying function.

However, we don’t actually apply the Bellman equation ourselves, but let the ANN do that: gradicent descent. (logisch, becasue we don’t have a table but a complex ann where we don’t know what to change) (L2 difference between our netresult and “target”, coming in part from reality and in part (maxaQs’a) from our net

Network itself is first CNN for visual stuff, followed by rectifiers [auch das geht mit selus mittlerweile besser!), then fc.

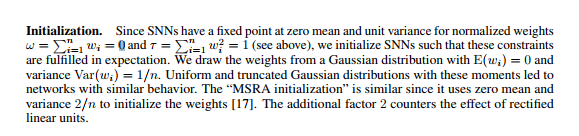
Also we clip error differencs & rewards

Erweiterungen: Batchnorm, Double-Q, …

Batchnorm: <https://standardfrancis.wordpress.com/2015/04/16/batch-normalization/>

SELU-Paper: *To robustly train very deep CNNs, batch normalization evolved into a standard to normalize neuron activations to zero mean and unit variance [20]. Layer normalization [2] also ensures zero mean and unit variance, while weight normalization [32] ensures zero mean and unit variance if in the previous layer the activations have zero mean and unit variance. However, training with normalization techniques is perturbed by stochastic gradient descent (SGD), stochastic regularization (like dropout), and the estimation of the normalization parameters. Both RNNs and CNNs can stabilize learning via weight sharing, therefore they are less prone to these perturbations. In contrast, FNNs trained with normalization techniques suffer from these perturbations and have high variance in the training error*

→ Falls ich SELUS mache:



...und anderes dropout

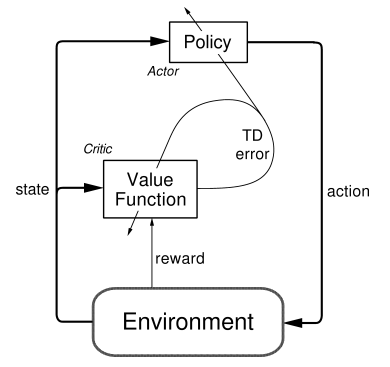
[ → es ist entweder selu oder batchnorm… und bei wenigen layers kein selu würde ich sagen….]

Auf zero mean und unit variance for normalized weights achten, auch wenn nicht selus!!!!!!!!!1111einself

[oookay, also quoten muss ich für das pure network: tensorflow, adam, batchnorm, double qlearning, dueling q-learning, selbst nur fürs DQN das ddpgpaper wegen den soft updates)

[policy gradient, reinforce, …: http://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html]

Actor-critic ist wieder was anderes



The critic is the value function, you can also think about this as the Bellman equation approximator. It returns Q values which describe the expected reward for an action. The critic maps a state/action pair to a single scalar value. This stands in contrast to Deep Q Networks (Mnih et al 2015), where the Q network maps the environment's state to a vector of Q values, one for each action. This is because in our case the Q network is not used to determine which action to take (DQN uses a deterministic argmax policy on the Q value vector), but only to *criticize* whatever action the actor network decides on taking. The critic is simply optimized by minimizing the mean squared error between the Q target values which are computed using the Bellman approximation on the experiences we make in the environment (we will do so below) and the actual output from the network. Critic receives action+state as input.

Actor networks describe the policy the agent should follow in any given state. Given some input the policy deterministically provides an action. Actions here are vectors of real values -- in a racing environment such a vector would for example contain a steering angle and a acceleration value. Actor recieves only states as input.

Our goal is to tweak the critic network's parameters such that it outputs action values, which, as input to the critic network, result in good results (*aka* high Q values *aka* profit): The actor is trained by ascending the gradients of the critic with respect to the actor's actions.

To do so, we need to perform gradient **ascent** on the critic's gradient with respect to the actor's action output (which is an input to the critic, a necessary condition for computing such gradient). That gradient, lets call it *value gradient*, depicts the direction in which to adjust the action (originally the output of the actor network) to minimize the Q values -- we ascent it, because we want to maximize the Q values.

Continous actions:

Discretizing the wheel into $5^\circ$ increments might work, but leads to a combinatorial explosion, as well as not allowing all the degrees of turn in between. This problem compounds and ultimately leads to the realization that Q-learning is insufficient for continuous actions because of the table representation. The table would need to be infinitely long in the *Action* axis, and to take the $\textrm{max}$ of an infinite series of numbers is impossible. On the bright side, continuous actions gives us a policy gradient, and when there's a gradient, there's the ability to optimize. In this case, maximization is what we want because we want to maximize our future reward. If we maximize this policy gradient, we will be able to learn continuous actions in reinforcement learning.

[....] we refer to @Sutton2000 for policy gradients in general, @Silver2014 for the deterministic policy gradient, and finally @Lillicrap2015 for deep deterministic policy gradients.

Policy-Gradient (PG) algorithms optimize a policy end-to-end by computing noisy estimates of the gradient of the expected reward of the policy and then updating the policy in the gradient direction. It has a stochastic policy giving a probability distrubition over actions, increasing the probability of making good actions.

In actor-critic-algorithms, the policy function (-actor) is independent of the value function (-critic). The actor produces an action given the current state of the environment, and the critic produces a TD (Temporal-Difference) error signal given the state and resultant reward. Since the critic estimates Q(s,a), it also needs the output of the actor. The output of the critic drives learning in both actor and critic.  
DDPG is a deterministic policy, which is much easier to learn.Policy gradient algorithms utilize a form of policy iteration: they evaluate the policy, and then follow the policy gradient to maximize performance. The input of the actor network is the current state, and the output is a single real value representing an action chosen from a **continuous** action space (whoa!). The critic’s output is simply the estimated Q-value of the current state and of the action given by the actor. The deterministic policy gradient theorem provides the update rule for the weights of the actor network. The critic network is updated from the gradients obtained from the TD error signal.

Okay. Actor-Critic-Methods consist of two different function-approximators, the actor and the critic. The critic is the Q-Value-estimator. It gets as input the State and the action and returns a single Value: Q(s,a).   
In the context of ANNs, using an online and a target version for each of these networks proved again useful. The online version gets updated every step, while the target network determines the directions in which the online networks are updated.

Since both algorithms are off-policy, it is easy to pretrian blablabla

Gym/Universe, Torcs, ….!!!!

Self-driving cars

* Wie realisitsch diese “ground truth” ist, ob das von LiDAR kommen kann
* Was NORMALERWEISE die steps sind: Sensing, localization, planning, action (? wie war das auf MIT self-driving cars präsi?)
* RRT\*

Ich und mein Auto

In the following part, I will write about the specific implementation of the blablabla.

The testing took place on a Windows 10 Machine <with the following specs and graka and blabalbla>. Using windows for testing was the best option, as its the only major platform on which both unity (available for win and mac), as well es tensorflow (available for win and linux) are available, and as network communication would slow down the response time of python a lot, windows is the only real option, suckers.

Da ich mich ja auch sehr sehr viel mit der connection unity-python beschäftigt habe, natürlich auch lange drauf eingehen. Überhaupt wie das mit rl ist, dass es halt den environment-state gibt und den state des agents, wie ich das mit dem server gelöst hab, wie daher der code aufgebaut ist, aber auch(!) wie das modernerweise heutzutage mit gym und universe gemacht wird

The actual game - what was given from Leon

* The original implementation of the racing simulation was implemented by the first supervisor of this thesis, Leon Sütfeld, and is available under the XYZ license. In its first version, the implementation featured the mode for manual driving, however no possibility to export the driven data for supervised learning, nor any possiblity for a ML-algorithm to take the wheel.
* Likewise, the first version of the visionvector as well as other vectors which will be the basis for python’s learning are already created, albeit in a less efficient version as of now (not Using Unitiy’s native renderers, lacking GPU performance)

What I did in order

* A fast communication between Unity and Python, which is natively able to run Tensorflow<source>, is an absolute necessity. For fast communication between different programming languages, Named Pipes [[1]](#footnote-0), zeromq or Sockets are considered good options, and because of me having to use windows, Sockets appear to be the smartest choice, as they are highly optimized under Windows, bypassing the non-necessary network layers when both parts of the connection are on the same system.[[2]](#footnote-1) After more and more optimizing and making both parts asynchronous, evaluations seems to proof this method right: The average time between Unity sending python its result and receiving a random result (without the TF inference) takes X ms on <machine>. Together with an average runtime of an inference in tensorflow taking Y ms, this leads to a maximum in FPS of 40 (notprovenyet), which is comparable to Googles DQN and Universe&GTA, being run in realtime, or 20FPS in twice the realtime. [mehr infos zu den sockets, dass das auf beiden seiten asynchron ist, dass das nen ziemllcih brachiales proprietäres protokoll ist, dass aber auf jeden Fall dafür sorgt dass die messages gesendet werden dass früher Unity nach pythons neuesten value gefragt hat, das aber quatsch ist)
* Natürlich auch wie das in Python dann funktioniert - mit senderconnnecter und receiverconnecterthread, die senderthread und receiverthread erstellen, aber alle ein gemeinsames, threadsafes, inputval-object haben, welches nötig ist wegen den historyframes, like it happened in DQN …. INFO B - ist das nicht quasi ne abgeänderte version vom observable-pattern? Mit der update()-methode?
* Also, principles of communication: PYTHON is server, connections-anfragen werden in asynchronem thread ausgelesen und ihr wert in der selben variable gespeichert, as soon as unity is connected, python waits for the result, handles it asap[in earlier versions unity asked for values, bad) in another thread, and sends (over another port) the result back to Unity. Unity, being the client, establishes the connection to python, sends the most useful results and recieves back from python, storing the result in a queue, and then taking the one which is exactly X=100ms seconds old
* Wie sieht dann python aus - es gibt die server.py die die kommunikation regelt blabla, und dann gibts nen agent der folgende funktionen beinhalten muss [...]. Memory war am anfang ineffizeint, ist aber mittlerweile optimiert towards unserem approach (2 kameras mit jeweils historyframes, …)
* Was für kommunikationsmöglichkeiten zwischen python und unity da sein müssen: hauptsächlich natürlich vision vs result, aber biede müssen sich auch sagen sollen dass sie sich resetten sollen (unity python bei wallhit, python unity wenn was passiert), und python muss unity pausierne können fürs lernen udn wenn es hängt
* Dass ich erst lernen und inference mit 2 threads parallel machen wollte, das aber die runtime ultra erhöht,. Dennoch sind aber learning und inference decoupled (das einzige was sie ja beide brauchen ist das replay memory, they are even separate networks, obviously!), sodass sie momentan nacheinander laufen, es aber auch easily möglich wäre das lernen auf ner andereren remote-instanz zu machen, which may be implemented in the future.aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

-dass diese art visionvector LIDAR-realistisch ist!

<uhm, darf ich Leons programm opensourcen?>

-darauf eingehen dass ich den code absichtlich erweiterbar gehalten habe für weitere theses

-Erst nur was das programm tun SOLL - was schon da ist (von Leon), und was das minimale und optimale Ziel der Arbeit ist (konzeptualisierung).

Dann technologien und paradigmen(!)

*“Im folgenden Abschnitt wird auf die technologische Grundlage des Projekts, die diesbezüglichen Entscheidungen und die ihnen zu Grunde liegenden Überlegungen eingegangen. Außerdem werden in der Umsetzung des Projekts verfolgte Paradigmen und ihre Abgrenzung zu ihren in der Industrie verbreiteten Alternativen erläutert.”*

*Nach einer allgemeinen Diskussion sicherheitsrelevanter Aspekte (Abschnitt 3.1) folgt eine Einleitung in JavaScript als die zentrale für dieses Projekt eingesetzte Programmiersprache (Abschnitt 3.2). Daraufhin wird die konkrete Applikationsarchitektur analysiert, in Server (Abschnitt 3.3) und Client (Abschnitt 3.4) unterteilt.*

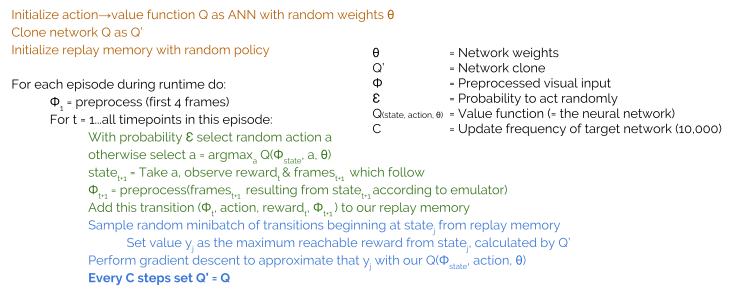
Alex hat das prinzip von https erklärt, mit dem satz am anfang “blablabla the following relies heavily on thisandthis source”

Alex nutzt auch (in fußnoten, of course) Quellen wie stackoverflow oder so, ergo darf ich den tweet von dingsi auch nutzen!!!!

Beim Eingehen darauf dass er javascript nutzt schreibt er auch mehr über die Frage warum erjs nutzt… (könnte ich drüber schreiben warum ich windows nutze, eh?) (“*Das Open Source Projekt Node.js (auch node genannt) wurde 2009 vorgestellt. Ryan Dahl begann die Entwicklung aufgrund seiner Frustration mit den zu diesem Zeitpunkt vorherrschenden WebserverLaufzeitumgebungen”)*

*(->ich kann auch tatsächlich auf den MS-deeloper-stackoverflowpost eingehen dass sockets maximal-schnell sind!) (überhaupt mal länger auf sockets eingehen mein gott!)*

In a first attempt, I strictly followed this approach:



What I noticed during implementation:

* Initializing the replay memory with a random policy is kinda useless, since the car does not get to any relevant points, and will alway stay close to where it started, never faster and a few KPH
* Random actions like done in the DQN paper are generally a bad idea in a racing game, since it is really really not a good idea to steer right being on a straight path at full speed
* For me, a state is a bit of history plus speed. In theory the agent can calulate some kind of trajectory from the history (speed alone wouldn’t be enough because of drifting)
* We have such a clear separation of application and learning, that we can perform them simultaenously on different devices!
* Dass er nicht stehen bleiben soll wenn er gas steht
* Of course, dqn is generally not the best idea, because not only curse of dimensionality, but also the fact that always only one value learns, even though close ones are also fine - experiment with also giving close ones positive reward?

Quote Alex: *“Außerdem wird beispielhaft auf eine sich bei der Entwicklung gezeigte Herausforderung und ihre Bewältigung eingegangen”*

Rückblick/Ausblick:

*Auf der Softwareseite werden so beispielsweise die zuvor getroffenen 1 einleitung Entscheidungen bezüglich der eingesetzten Werkzeuge auf Grundlage der neu gesammelten Erfahrungen diskutiert*

Dass das gucken von correct inferences wie ich es am ende vom runnen des cnn mache kein guter weg ist die quality zu assessen, dass das überrepräsentieren von kurven schon besser wäere, dass aber am besten sowas wie ein vergleich mit RRT\* am besten wäre.. Nur braucht man dabei underlying physics

Continuous control paper

Actor-critic, model-free, off-policy [DQN is off-policy too, because of the replay buffer… and because DDPG is off-policy, the replaybuffer can be large, allowing the alrogithm to benefit from uncorrelated transisions], offline [not using minibatches & minibatches wäre angelbich online?]

Dqn can handle highdim observation spaces, but only discrete, lowdim action space. DQN cannot straight-forward handle continous stuff, since it relies on finding the action maximizing the action-value function, which in the cont-val case requires iterative optimization process at every step. Discretizing leads to the curse of dimensionality. Large action spaces are difficult to explore efficiently, sucessfully training DQN-like networks is likely intractable, and we’d lose necessary information anyway. The approach only requires a straightforward actor-critic architecture. Discrete Timesteps, typical markov decision process. Actions are real-valued. Assumes fully-observed environment. (!!).

What an agent generally does: behaviours is defined by policy pi mapping states to probdists over actions. Return from a state depends on the actions, thus on the policy, and may be stochastic. What we also need here is Q, the action-value-function (describing the exptedted return after taking an action a in state s and thereafter following our pi). Then RL/Q-learning normally uses the bellman equation, which, if the target policy is deterministic (lets call the function müh then), boils down to a function Qmüh depending only on the environment, and can thus be learned off-policy, using transitions which are generated from a different stochastic behaviour policy (WHICH IS THE REASON warum ich auf human data pretrainen kann!) Soo, the general off-policy algorithm is Q-learning, using the greedy policy müh(s) = argmax(a)(Q(s,a)). Minimizes thetemporal differnece error.

t is not possible to straightforwardly apply Q-learning to continuous action spaces, because in continuous spaces finding the greedy policy requires an optimization of at at every timestep; this optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action spaces. Instead, here we used an actor-critic approach based on the DPG algorithm (Silver et al., 2014).

The actor == the policy. It is deterministic, mapping state to action. The critic is used learning the bellman equation as in Q-learning. Used soft target updates, used replay buffer, used batchnorm… used this Ornstein-Uhlenbeck-Process for temporally correlated exploration similar, but not equal to Wawryzynski

Dass ihr TORCS-result war: “On both low-dimensional and from pixels, some

replicas were able to learn reasonable policies that are able to complete a circuit around the track though other replicas failed to learn a sensible policy.”

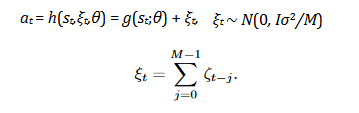
Exploration

The original Deep-Q-Network was a policy-based Network. As such, the only way to implement exploration is by performing random actions. (well, das ding ist dass man \*eigentlich\* random actions, corresponding to button-presses, nicht nem random-wert hinzufügen kann weil das semantisch keinen sinn macht. Daher ist die einzige wahl hin und wieder ne random action zu machen. Bei meinem approach ist das natürlich wieder anders, ich kann auch beim DQN nen gewissen noise hinzufügen) While DQN used the easiest of those approaches (epsilon-greedy), there are several other possibilities of implementing exploration. One follow-up paper [https://arxiv.org/pdf/1606.01868.pdf] for example made a pseudo-count, which roughly counted how often the agent already performed a random action in this state, such that there is no global epsilon, but every state has its own probability for a random action, decreasing with the implemented pseudo-count of already performed actions. Using this approach, they performed significantly better on games which are perceptually highly aliased whilst having very sparse rewards, like the atari-game montezumas revenge.

[jetzt auf andere exploration-strategies eingehen, see this]

[was macht überhaupt jetzt den unterschied aus zwischen off-policy und on-policy?? Ist dqn off-policy??]

Performing random actions in a racing game is obviously catastrophic. Driving maximum speed, and then suddenly steering to one side as hard as possible will inevitably lead to a fatal crash. [similar problems as in robotics, see this <http://www.ijmlc.org/vol5/489-A16.pdf>] Luckily, there are other possibilities of how to perform exploration, when the meaning of the network-outputs are known. In our case, one can let the network do its thing, and then add noise to the suggested outputs of the network [following a uhlenberg-dingsi-process [quote], as quoted in [DDPG].

Quote <http://www.ijmlc.org/vol5/489-A16.pdf>: In direct application of RL in robotics you have a discontinous control signal - consecutive actions are selected independently on random, often making them excessively far from another, which in robots may lead to their distruction. So, they consider a control policy in which consecutive actions are modified by autocorrelated noice. They used actor-critic networks btw. They have the problem, that during learning, the robot’s joint positions are selected at random, making the joints jerk, which is unhealthy behaviours für robots. Of course, for cars its even more fatal, blabla. They are trying to get a control policy for general RL algorithms that does not make the robot jerk, which is exactly what I need. Their policy bases on deterministic transformation of state combined with a random element, which is the stochastic process of the moving average. Both actions and states are multidimensional cintinous. A timestep st reflects the state of a certain continuous-time system at discrete time-instants [formula]. They also assume a high-level learner (aka a neural network), telling the joints where to be and a low-level mover, interpolating between those positions. What is wanted here is, that even fine time discretization doesnt result in jerking. (yes, I want the same). Because of that, an action is produced by a function taking into account the actual neuralnetfunction, state and some random process(xi ist random element, theta ist weighs, s ist state). OTHERS ASSUME that xi has the same distribution for various t, and xi(t) is independent of xi(t+i), i größer 0. THEY assume xi has the same distribution for each t, that xi(t) is stochasically independent from xi(t+i) from a certain i größer 0 on, something else, and h is continuous. The last two assumptions make consecutive control acions close to another. (what they say is that xi(t) is on average close to xi(t+1). So they select random actions with 

Whereas the lower one is the moving average. That fulfills their conditions trivially (moving average undso). The distribution of action defined this way is normal. What they still need afterwards is a parametrization of that control policy that assures constant level of randomness for any given time discretization. [mathmathmath]...One thing he showed is, that if M, length of moving average, is 1, then his stuff is equal to the normal RL control policy. That, as he showed [achtung, main to-quote-thing!], is bad as the variance of noise required to stabilize the power is inversely proportional to time discretization - in highly discretized, then the variance must be very large - that requires large changes in control signal performed in large frequency - jerking movement, sucks. Sooo instead, we use the ones with autocorrelation, where random movements are similar to random movements from before, that rules [smaller time-to-time differnces in control signal]. Another thing he proposes is that the weight oof a reward received in a specific time in the future should remain the same, even though its received after a different amount of discrete timesteps, the discount factor should thus be modified accordingly [formula 28]. [schreiben dass, wenn man die weights verändert, man sich an seine formulas hier halten soll!!]

Soo, inorder to learn, algorithms need to perform random actions, collecting experience, however for these actions to make a differnce, they need appropriate amont of randomness, and the finde rtime discretization, the less relevant an action i, the more noise it should contian, the less continuous an control signal is, that is ver yinappropriate. Thats why we want autocorrelated noise.

* Hierdrauf verweisen für wenn man discretization ändern will
* Das problem mit dem jerking ist bei mir ganz ähnlich, moving average als lösung
* Überhaupt, nicht random actions machen, sondenr die gegebene action mit nem gewissem faktor verändern
* He also mentions the problem, that the coarser the time discretization and independently seleced actions, it is easy to verify quality of each action, making them more or less likely.

Quote <https://arxiv.org/pdf/1606.01868.pdf> (Unifying Count-Based Exploration and Intrinsic Motivation)  
Exploration in non-tabular RL. Using density models to measure uncertainty, propose an algorithm for deriving a psuedo-count. Generalizes count-based exploration algorithms to the non-tabular case (yes, seems to be the way I had it). They make pseudo-count from raw pixels, transform that into explortion bonuses.

In tabular settings, confidence intervals and posterior shrink as the inverse sqrt of the state-action visit count. There are count-based exploration algoritms, that give an exploration bonus for actions not oftne visited (proportional to the count of each state-action pair. That makes the agents suboptimality finitely-timed. Can easily be seen that that provides theoretical guarantees. Not applied in DQN yet however, because no visit-counts possible.

Intrinsic motivation is something else. In Intrinsic motivation, the agent wants to visit states where the prediciton error (learning process) is high. This also works in non-tabular algorithms.  
They show that those two are the same. The pseudo-counts they provide are “function approximation for exploration”.

Reward

Im haben die reward corresponding to speed in street-direction, however as can be seen in [the screenshots], that leads to real quick stagnation in a local optimum, especially because the first curve is that hard. As can be seen in this screenshot, high rewards do not correspond to rounds where the agent made a lot of progress. Also, when testing that agent, it turned out that it didn’t brake at all.

A good reward corresponds to far progress [naja well, am anfang, am ende soll er diesen progress auch in kurzer zeit machen]. Problem of that is however the typical reinforcmentlearning problem, that a single action doesn’t know how much it contributed to the overall progress.

Die Idee dass wir nach nem reward suchen, bei dem die runden mit guter zeit [anfangs clearly nur die menschlichen runden] zu jedem streckenabschnitt hohe werte zeigen, und schlehcte niedrige.

Performance measure

Simply counting how often the agents performs the same action as a human has of course several disadvantages. First of all, this assumes that the human actions were always a perfect ground truth, which can’t be taken for granted. Secondly, in the non-continuous agents, you lose so much information by discretizing the action, such that it is very conceivable that an agent, that discretizedly performs the exact same acitons as a human, drives into a wall asap, while a human makes a perfect drive. Another aspect is, that a huge part of the game actions in the game consists of simply steering forwardly, and thus an agent simply steering forwardly already achieves quite a good accuracy using this measure. [errechnen wie viel prozent accuracy ein agent hat der die ganze zeit nur geradeaus-gas-gibt], while obviously being far from a good agent. Natürlich könnte man das als baseline nehmen, blablabla.

-dass die beste wäre das mit dem RRT\* zu vergleichen

The game (Unity-part)

* Screenshot vom game
* Welche inputs gibt es
* Was wird python warum gesendet (auch dass visionvec monocrhom ist)
* Das mit dem wie realistisch es ist unseren visionvec zu haben (LIDAR?)
* Das mit den sockets
* Dass es nicht deterministisch ist (zeigen mit nem plot?)
* Was leon getan hat, was ich getan habe
* Dass es live ist, dass man eingreifen kann und sich q-werte plotten lassen kann
* Dass es supervised runden exportieren kann
* Diagramm von der Kommunikation

The learner/agent

* Unbedingt nen UML-diagramm
* Dass der bei 0kmh beschleunigen muss
* VERGLEICH mit vision-vector und mit nur den othervecs(!)

The network(s)

Dueling, Double, blablabla (im DDPG-paper steht auf seite 6: Qlearning is prone to oeverestimating values [Hasselt, 2010])

Dass [siehe medium-post] das action-repeat gar nicht so gut ist, und recurrent besser wäre

1. <https://stackoverflow.com/questions/1749001/named-pipes-between-c-sharp-and-python> Accessed Donnerstag, ‎6. ‎Juli ‎2017 [↑](#footnote-ref-0)
2. According to former Microsoft developer Jeff Tucker on Stackoverflow: <https://stackoverflow.com/questions/10872557/how-slow-are-tcp-sockets-compared-to-named-pipes-on-windows-for-localhost-ipc> [↑](#footnote-ref-1)