UNCERT: A PyTorch Based Implementation of Semi-Model-Based Reinforcement Learning with Uncertainty

Simon Lund, Sophia Sigethy, Georg Staber, and Malte Wilhelm

Ludwig Maximilian University of Munich
{simon.lund,s.sigethy,georg.staber,malte.wilhelm}@campus.lmu.de

Abstract. We present a probabilistic neural network for modeling the system dynamics of the Cartpole environment in OpenAI gym. This network replaces the default environment and is able to express uncertainty, regarding its predictions. The Bayesian Neural Network is used to mimic an unreliable environment which a Reinforcement Learning agent tries to learn. We show that the RL policy uses strategies to avoid areas with high degree of uncertainty.

Keywords: OpenAI Gym · Cartpole · Reinforcement Learning · PyTorch · TensorFlow · Probabilistic Bayesian Neural Networks · Uncertainty · Proximal Policy Optimization

1 Introduction

Reinforcement learning (RL) represents one of the most high-profile research areas in the context of machine learning and the field of general artificial intelligence. Using games as a proxy, prominent papers like the Alpha-Algorithms (AlphaGo [22], AlphaGoZero [24], AlphaZero [23], AlphaStar [29]) have significantly contributed to the advancements of general AI competency. By starting with games, researchers are able to translate the learnings into the real world. Recent publications, such as AlphaFold [11], have shown the potential of RL applications, especially when combined with deep learning techniques.

However, compared to simplified computer simulations, real life data is often noisy, scarce and unreliable. This can cause issues because the prediction quality of neural networks is highly dependent on the quality of the training data. So, with lack of information, the modelling of the interested function becomes *uncertain* regarding the accurate correlation between inputs x and outputs y. Preferably, an agent is able to evaluate its prediction in terms of uncertainty [8]. The resulting model could be applied to classification tasks in medical radiography [16] or be used in active learning [15].

Yet, a comprehensive analysis regarding the implications and inner workings of uncertainty in the RL domain remains a desideratum of research. To the best

of our knowledge, we are the first to apply uncertainty to the OpenAI version of the CartPole environment.[1]

In this paper, we'll first lay out the most important terminology in section 2 Disambiguation and move on to an explanation of the concept in section 3 Concept. After this, we'll discuss the experimental setup and evaluate the results in section 5 Experiments and section 6 Evaluation, respectively. We will end this publication with the Conclusion in section 7.

2 Disambiguation

2.1 Reinforcement Learning

Reinforcement Learning (RL) [26] is one of the three pillars of machine learning. As opposed to supervised learning and unsupervised learning, RL is a model-free approach driven by experience in order to solve problems autonomously. The agent starts tabula rasa, meaning it does not have any comprehension a priori and learns through experimentation.

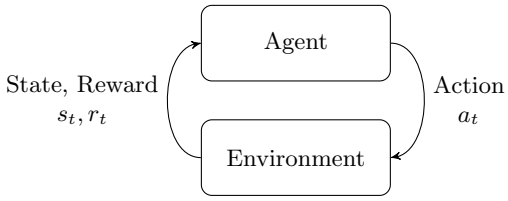

Fig. 1: Agent-environment interaction loop.

RL consists of two main aspects: the *agent* and the *environment*. For every iteration of the agent-environment interaction loop, the agent observes a state of the world around him, which might be incomplete. The environment may change according to the action taken by the agent or on its own. In basic RL it is usually assumed that the environments fulfills the Markov property and is thus formalized as a Markov Decision Process (MDP), which can be expressed as a quintuple (S, A, R, P, ρ_0) [30]. S and A denote the set of all valid states and actions, respectively. The reward function is represented by $R : S \times A \times S \to \mathbb{R}$, with $r_t = R(s_t, a_t, s_{t+1})$, referring to the reward gained by taking action a_t to move from state s_t to state s_{t+1}. The probability of transitioning from state s to state s' with action a and a starting state distribution of ρ_0 is given by $P(s'|s, a)$. This is part of the transition probability $P : S \times A \to \mathcal{P}(S)$. As seen in Figure 1 the agent starts from a state $s_t \in S$ and picks an action $a_t \in A$

[1] For the sake of completeness, it should be mentioned that Moberg [18] has already compared methods for generating uncertainty-aware neural networks with the CartPole Environment but in the MuJoCo framework [28].

according to a policy. It then receives a new state $s_t \in S$ as well as a reward r_t and can begin a new iteration. The overall goal in RL is to select a policy so that the cumulative reward is maximized. In a stochastic environment, this can be formalized as seen in Equation 1 [2].

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{1}$$

$$J(\pi) = \int_{\tau} P(\tau|\pi) R(\tau) = \mathop{\mathrm{E}}_{\tau\sim\pi}[R(\tau)] \tag{2}$$

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\, \pi(a_t|s_t) \tag{3}$$

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{4}$$

This policy π^*, also called optimal policy, maximizes the expected return (Equation 2), so the integral of the probability of a T-step trajectory (Equation 3) multiplied with the infinite-horizon return discounted by a factor of $\gamma \in (0, 1)$ (Equation 4). Within RL many types of algorithm exist (see Figure 2). Model-free approaches can be divided into Policy optimization and Q-Learning.

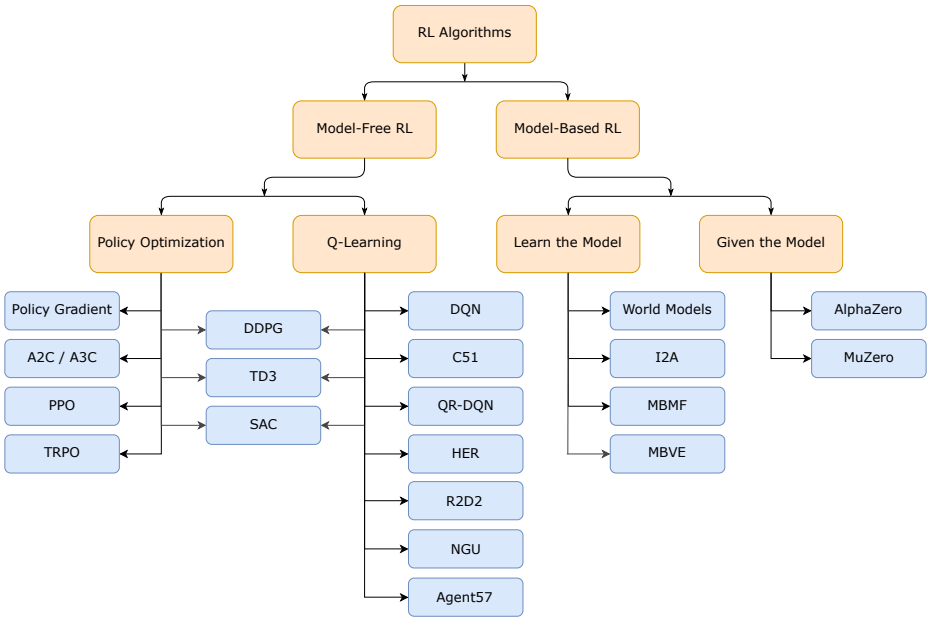

Fig. 2: An updated taxonomy of RL Algorithms based on Achiam's *Spinning Up in Deep RL* [2].

2.2 Model-Free RL

Q-Learning In order to approximate the Q-function, the expected value of the sum of future rewards by following policy π, the Q-Learning algorithm can be deployed. The idea behind this being, a decent approximate of the Q-function for all state and action 2-tupel, should return the optimal policy π^* through the arguments of the maxima. To train the agent to output the true Q-value at a particular instance or several states, respectively, one needs to maximize the target return over an infinite time horizon to serve as ground truth. To facilitate this training, the *Bellman equation* is used iteratively to calculate the mean squared error (or Q-Loss) of the target value and the predicted value (see Equation 5).

$$\mathcal{L} = \mathbb{E}\left[\left\|\left(r + \gamma \max_{a'} Q\left(s', a'\right)\right) - Q(s, a)\right\|^2\right] \tag{5}$$

Using Deep Neural Network to approximate the Q-function is known as Deep Q Learning (DQN) [17].

Policy Gradients The main idea of policy gradients is to output policies directly without the need for an intermediate Q-function. This is done by increasing the probabilities of actions which yield higher returns and decreasing the likelihood of actions that result in lower returns. This is repeated until the optimal policy is reached. The key equation for this is pictured in Equation 6.

$$\nabla_\theta J\left(\pi_\theta\right) = \underset{\tau \sim \pi_\theta}{\mathrm{E}}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(a_t | s_t\right) A^{\pi_\theta}\left(s_t, a_t\right)\right] \tag{6}$$

π_θ denotes a policy with parameters θ, and $J(\pi_\theta)$ represents the expected finite-horizon undiscounted return of the policy. τ is a trajectory and A^{π_θ} denotes the advantage function for the current policy, so the extra reward if this action is taken. Policy parameters are updated via stochastic gradient ascent on policy performance: $\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k})$. This means for a good action a with a large reward the likelihood will be increased and for a bad action (small or negative reward) the probability will be decreased.

2.3 Proximal Policy Optimization (PPO)

PPO [21] is the de facto standard RL algorithm within OpenAI due to its simple usage and its ability to reach good performances. It is comparable or superior to state-of-the-art algorithms, while being easier to tune and easier to implement. The main concept of PPO is to calculate an update at every single step that minimizes the cost function while guaranteeing that the deviation from the previous policy stays rather small. The objective function is as follows:

$$L^{CLIP}(\theta) = \hat{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \operatorname{clip}\left(r_t(\theta), 1-\varepsilon, 1+\varepsilon\right)\hat{A}_t\right)\right] \tag{7}$$

Hereby, θ denotes the policy parameter, $\hat{E}_t$ the empirical expectation over time steps, $\hat{A}_t$ the estimated achieved advantage at time step t, r_t the probabilistic ratio under the new and old policy and ε the learning rate. This objective function relies on a method to do a Trust Region update that is equivalent to Stochastic Gradient Descent, and it simplifies the method by removing the Kullback–Leibler divergence penalty and the requirement to make adaptive updates.

2.4 Uncertainty Quantification

Not all uncertainty is created equal, and it may be quantified into two different categories [13]. Figure 3 showcases the different sources of uncertainty (*aleatoric* and *epistemic*).

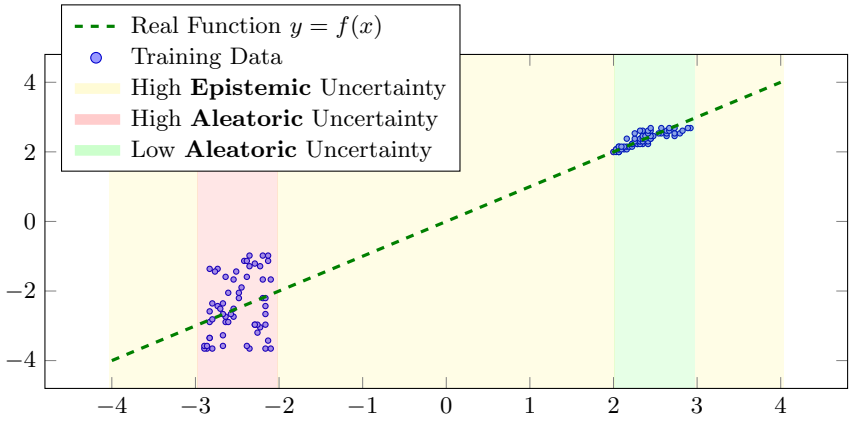

Fig. 3: Uncertainty Quantification. As the aleatoric uncertainty differs across the inputs, the data set is also called *heteroscedastic*.

Aleatoric Uncertainty Aleatoric, derived from the Latin *aleatorius* "belonging to the player", *alea* "dice, risk, chance", refers to uncertainty which is inherent to the environment. This is the statistical uncertainty, which is present regardless of the number of repetitions and the amount of data. This kind of uncertainty is irreducible, as it is part of the environment itself. Figure 3 shows an exemplary data set in which the values $x \in]-3, -2[$ are characterized by a high degree of aleatoric uncertainty (red section) because they offer a high variance to the real function $y = f(x)$. Opposed to this, the samples $x \in [2, 3]$ are comprised of a low aleatoric uncertainty, as the standard deviation around the original green function is low.

Epistemic Uncertainty Epistemic uncertainty, from Ancient Greek episteme for "science, knowledge", incorporates approximation uncertainty as well as

model uncertainty, and refers to uncertainty ascribable to lack of knowledge about the environment [10]. This type of uncertainty can theoretically be reduced. An example of epistemic uncertainty can be seen in the yellow shaded portion of Figure 3 as there is no training data available.

2.5 Probabilistic Deep Learning

To introduce probabilistic deep learning, let's recall a traditional multiclass classification problem: For a given input image x we want to find a neural network, which will assign class probabilities $\hat{y}$ as close to the ground truth y as possible. $||\hat{y}||$, the length of $\hat{y}$, is the number of output classes (e.g. 3 - cat, dog, bird). This will work well, if the input image actually matches any of the known output classes, however, if we provide the neural network with something novel (like a boat) it will still try to assign it to one of the known classes. So it will still try to represent a conditional probability distribution $p(y|x, \theta)$ with θ being the parameters (point weights) of the network. We can see this in Figure 4a were the model has been able to recognize the overall tendency of the data samples (blue dots) using the overall mean. However, the training data is heteroscedastic, as x increases y get more variable, and the neural network is unable to accurately detect this.

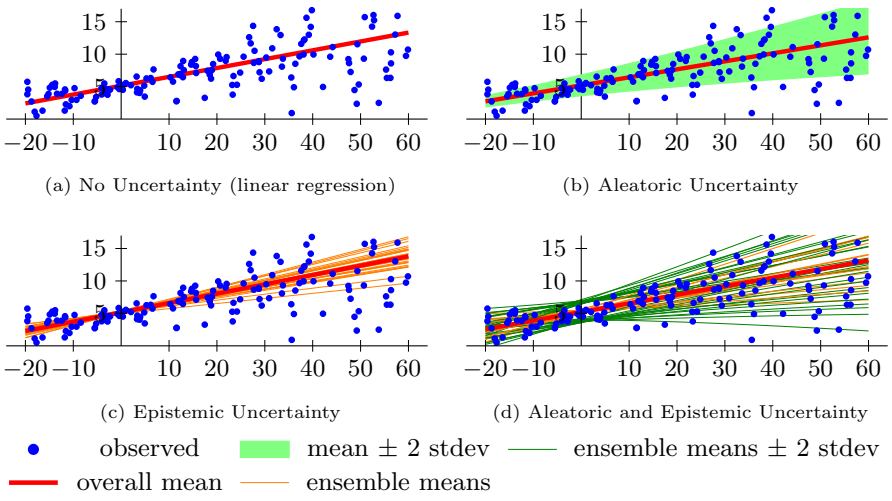

Fig. 4: Model evolution to handle an increased amount of uncertainty, in ascending order from a to d. Figures from the TensorFlow Blog [25].

So, it would be useful to associate each predication with an uncertainty. Then, the boat would still be assigned a label like cat, but with a high uncertainty. To achieve this, the weight of the neural network need to be parameterized

using probability distributions over possible values instead of point weights, also known as a *Bayesian Neural Network* [5] (see Figure 5). The model is now less certain what y is supposed to be as x grows larger (see Figure 4b). To also cover epistemic uncertainty we start with some belief in the form of the prior distribution, after we observe the actual data, we update our prior assumptions accordingly, to calculate the posterior distribution (see a visual sketch of Bayes' rule in Figure 6). In the next iteration, the posterior distribution acts as the next iteration's prior distribution.

So we end up with an ensemble of models with different parameters θ, and by plotting their average weighted by the posterior probabilities of their parameters we can see the uncertainty of the model (see Figure 4c).

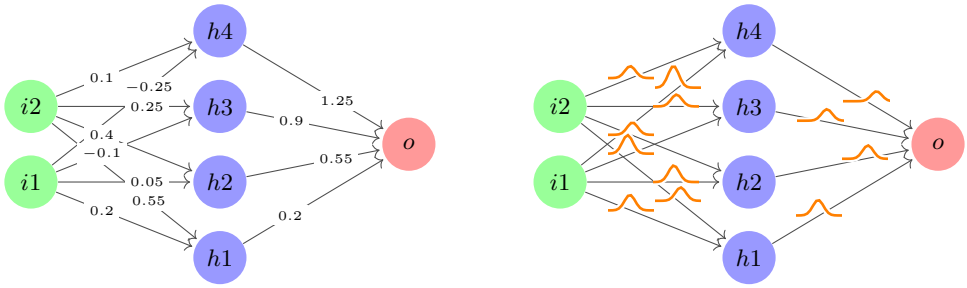

Fig. 5: A comparison between Convolutional Neural Network and Bayesian Neural Network. Figure adapted from Blundell et al. [5] and cited from LaTeX StackExchange [1].

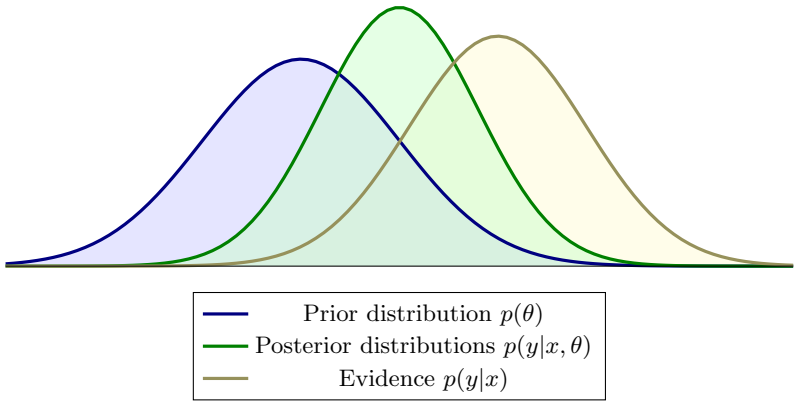

Fig. 6: Probability distributions entangled by Bayes' rule. Illustration based on Kamperis [12].

2.6 Lossfunction: Mean Squared Error

The average of the squared differences between actual and anticipated values is computed by the Mean Squared Error (MSE), also known as L2 Loss. A lower L2 value suggests a higher accuracy of the model — an ideal value is 0.0. Due to the squaring, larger inaccuracies result in much larger errors than smaller ones. The error is 10,000 if the classifier is wrong by 100. The error is 0.01 if it's off by 0.1. This penalizes the model for major errors while rewarding minor ones.

$$\text{loss}(x, y) = (x - y)^2 \tag{8}$$

3 Concept

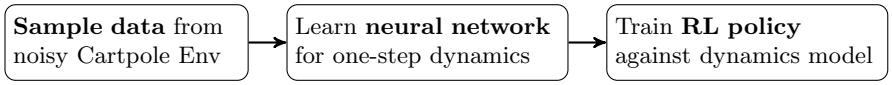

Fig. 7: The workflow used to facilitate semi-model-based RL with uncertainty.

CartPole is an inverted pendulum, this means the balance point, also known as center of mass m, is above its pivot point p. Once θ, the deviation of m from p, becomes too large, the CartPole would tip over. However, this can be prevented by moving the pivot point back under the center of mass. The challenge is to apply, the appropriate forces $\vec{F}$ at the correct time to the pivot point, in this case the cart M, in order to keep the CartPole balanced.

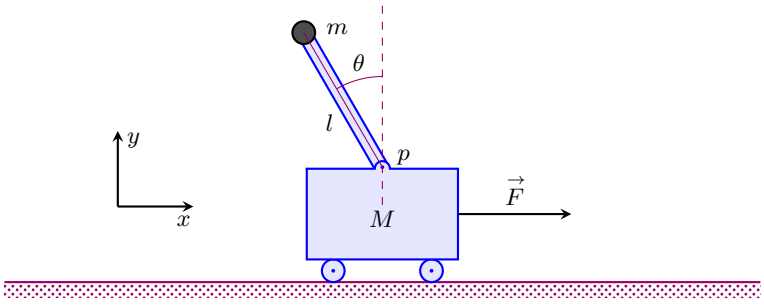

Fig. 8: A schematic drawing of the inverted pendulum on a cart adapted from Benmiloud [4]. M refers to the mass point of the cart, m denotes the mass of the pendulum, l refers to the length of the rod, θ is the pendulum angle, x and y are the horizontal and vertical coordinates of the cart respectively p denotes the pivot point of the pendulum and $\vec{F}$ is the force applied to the cart.

The CartPole environment has been a classic challenge in the RL domain and serves as an entry level problem nowadays. One of the most popular means to develop and compare reinforcement learning algorithms is to use OpenAI Gym [7], which is an easy-to-use benchmark environment written in Python. The toolkit aims to standardize environments for AI research publications. The CartPole environment is analogue to the version of the cart-pole problem outlined by Barto, Sutton, and Anderson [3]. In this paper, we use OpenAI Gym to model a noisy version of the CartPole environment with a probabilistic neural network in PyTorch [19].[2] This network is then passed to a RL algorithm of Stable Baselines3 (SB3) [20] to learn the uncertainty of the environment (see Figure 7). SB3 is a library which offers reliable implementations of established RL algorithms in PyTorch. We call our approach **semi-model-based**, as the neural net influences the learning process of the agent.

3.1 Data Sampling

First, we sample data on a fork of the OpenAI-Gym CartPole simulation [14]. This fork starts with the pendulum at the bottom. So in order to succeed, the agent need to swing up the pole. We adapted the simulation to create deliberately noisy data in certain parts of the state space (e.g. for CartPole angular position 130° - 230° and $2.3 - 4$ rad, respectively. It would also be possible to apply noise to part of the track, e.g.$x \in [-1, 0.5]$). Figure 9 shows this region as θ in beige. We realize the error by adding an error term ϵ to the true angle, where ϵ is normally distributed with mean 0 and variance c^2.

Expressing an angle in radians or degrees poses an additional challenge for machine learning algorithms, as they are not continuous functions. Therefore, the angle of the pole α is expressed in terms of trigonometric functions, more specific sine and cosine. These functions are continuous and have a periodicity of 2π. Another advantage is that these values are inherently normalized because the range of $cos(\alpha)$ and $sin(\alpha)$ is $[-1, 1]$.

3.2 Neural Network

The creation of the neural network is a three-step process. First, the architecture of the model is configured, then the network is equipped with an appropriate loss function and dedicated metrics. Next, the model is trained on the sampled data from the noisy CartPole environment. The specialty of the architecture is, that we use a neural network which optimizes a distribution. We pass observations for the last four time steps ($t_{\text{-}1}, t_{\text{-}2}, t_{\text{-}3}, t_{\text{-}4}$) as well as the action taken at the last time step $a_{t_{\text{-}1}}$. Every observation includes the x position of the cart, its velocity, the angle of the pendulum θ and its angular velocity. The output of the network given a sample is therefore not only a predicted value, but a distribution fitted to this value. In our case, this is a normal distribution with mean μ and variance

[2] The source code is available at https://github.com/github-throwaway/ARL-Model-RL-Unsicherheit

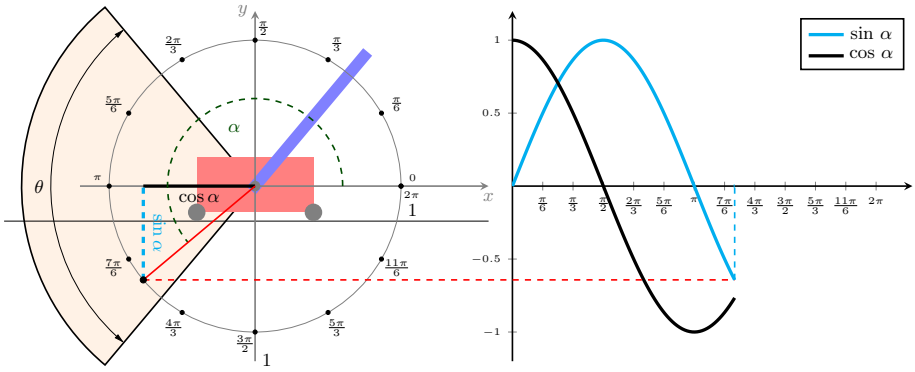

Fig. 9: This figure shows the CartPole superimposed with the unit circle. The actual environment has the circle turned 90° anti-clockwise. A circular segment denoted with θ and colored in beige represents the noisy section. If the pole (in blue) is within this interval, the environment returns intentionally unreliable data. The angle of the pole α can be expressed in terms of trigonometric functions, more specific sine and cosine. Figure adapted from Taskjelle [27].

σ^2, where μ describes the expected output and σ^2 represents the uncertainty of the prediction. As common for fitting distributions in statistics, the Mean squared error is used here as loss function (see Equation 8).

3.3 RL Policy

We use our neural dynamics model to train a reinforcement learning policy. A policy represents the behavioral strategy of a reinforcement learning agent. We encapsulate the dynamics model as an OpenAI gym environment and deploy different learning algorithms from Stable Baselines3 such as PPO2. The uncertainty outputted by the BNN is used deliberately during learning, e.g. as a cost in the reward function

4 Implementation

There are two slightly different implementation of this project. One in PyTorch and TensorFlow. The main one we refer to throughout this paper is written using PyTorch.

4.1 PyTorch

We use BLiTZ [9], a library to create Bayesian Neural Network Layers on top of PyTorch, to facilitate our project.

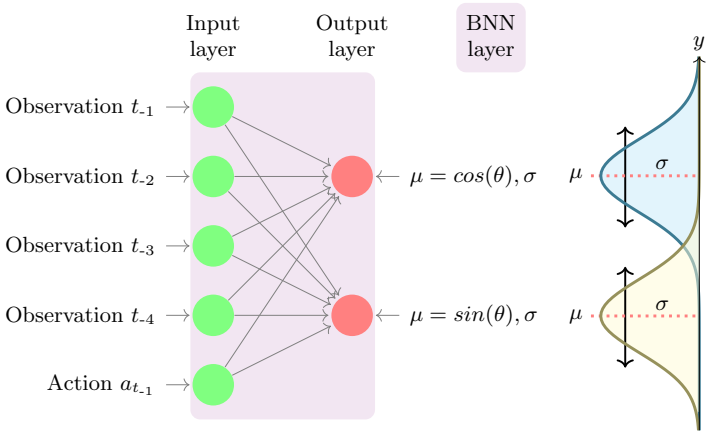

Fig. 10: Schematic architecture of our Bayesian Neural Network (BNN). It has 21 input nodes (5 oberservation parameters $*$ 4 time steps $+$ 1 action $=$ 21) and two outputs. The action $a_{t_{-1}}$ refers to the action which will take the agent from observation t_{-1} to observation t_0.

Creating a variational regressor class As with any Torch network, we construct our class by inheriting from `nn.Module`. In order to sample the loss, the `@variational_estimator` decorator by BLiTZ provides ways to handle Bayesian features, such as computing the complexity cost of the Bayesian Layers and performing multiple feed forwards (sampling various weights on each forward pass).

```
@variational_estimator
class BayesianRegressor(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        self.blinear = BayesianLinear(input_dim, output_dim)

    def forward(self, x):
        return self.blinear(x)
```

Defining a confidence interval evaluating function This function computes a confidence interval for each prediction in the batch from which we are attempting to sample the label value. The accuracy of our predictions may then be measured by determining how much of the prediction distributions really included the right label for the data point.

```
def evaluate_regression(regressor, x, y, samples=25, std_multiplier=2):
    preds = [regressor(x) for i in range(samples)]
    preds = torch.stack(preds)
```

```
    means = preds.mean(axis=0)
    stds = preds.std(axis=0)
    ci_upper = means + (std_multiplier * stds)
    ci_lower = means - (std_multiplier * stds)
    ic_acc = (ci_lower <= y) * (ci_upper >= y)
    ic_acc = ic_acc.float().mean()
    return ic_acc, (ci_upper >= y).float().mean(), (ci_lower <=
    ↪  y).float().mean()
```

Creating our regressor and loading data Next, we set up our optimizer with a default learning rate of 0.01 and MSE as our loss function. Note that in the original publication [5], the negative log-likelihood term in the total loss function is a sum over samples. So we scale the KL term `complexity_cost_weight` because the default `criterion` is the average.

```
optimizer = optim.Adam(regressor.parameters(), lr=0.01)
criterion = torch.nn.MSELoss()
complexity_cost_weight = 1. / x_train.shape[0]
```

Main training and evaluating loop The Evidence Lower Bound (ELBO) Loss is sampled for a batch of data consisting of inputs and corresponding-by-index labels. The ELBO Loss is the sum of the model's KL Divergence and the actual criterion - (loss function) of optimization of our model (the performance part of the loss). Because we're using variational inference, it takes several Monte-Carlo samples of the weights (quantified by the parameter `sample_nbr`) to get a better approximation for the loss.

```
losses = []
for epoch in tqdm(range(100)):
    new_epoch = True
    for i, (datapoints, labels) in enumerate(dataloader_train):
        optimizer.zero_grad()

        loss = regressor.sample_elbo(
            inputs=datapoints,
            labels=labels,
            criterion=criterion,
            sample_nbr=3,
            complexity_cost_weight=complexity_cost_weight
        )

        loss.backward()
        optimizer.step()
```

4.2 Usage of Argument Parser

For the sake of usability, we implemented an argument parser. By passing some predefined arguments to the python program call, it is possible to start different routines and also change hyperparameters needed by the algorithms. This enables the user to run multiple tests with different values without making alterations to the code. This is especially helpful when fine-tuning hyperparameters for reinforcement learning algorithms, like PPO. To get an overview of all the possible arguments, and how these arguments can be used, the user may call:

```
$ python main.py --help
```

4.3 TensorFlow

We also implemented a version of this project using TensorFlow. For this version, we employed `DenseVariational` and `DistributionLambda`. The detailed process is outlined in a blog post by the TensorFlow probability team [25]. However, we were unable to output the angle in terms of sine and cosine for this network. Therefore, we used the discontinuous function radian to express the angle of the pendulum. This caused problems in the learning process and decreased the quality of the learned RL policies significantly.

The post uses the Keras Sequential API, but this type of model is not appropriate when any of the layers has multiple inputs or multiple output. We also tried a different approach using the functional API, but this also did not lead to any significant success. Nevertheless, the code is included in the repository for the sake of completeness.

5 Experiments

To be able to make assumptions about the performance, we have to compare our implementation of the cartpole environment to the original cartpole environment, ours is based on. Therefore, we train a reinforcement-learning agent on it and compare it to another agent that was trained on the original cartpole environment.

5.1 Experimental setup

We use three differently trained variants of our environment to train an agent, using the PPO algorithm, which uses the same hyperparameters. The difference between the two plots is shown by coloring the area between them. The smaller the area between the original and the observed data, the better the model.

Therefore, we have to create different instances (varying in the amount of training data used) of our implementation, which behave as follows in comparison to the original environment. Since the observations in blitz50k were closest to the original env, we decided to only use this one for further investigations.

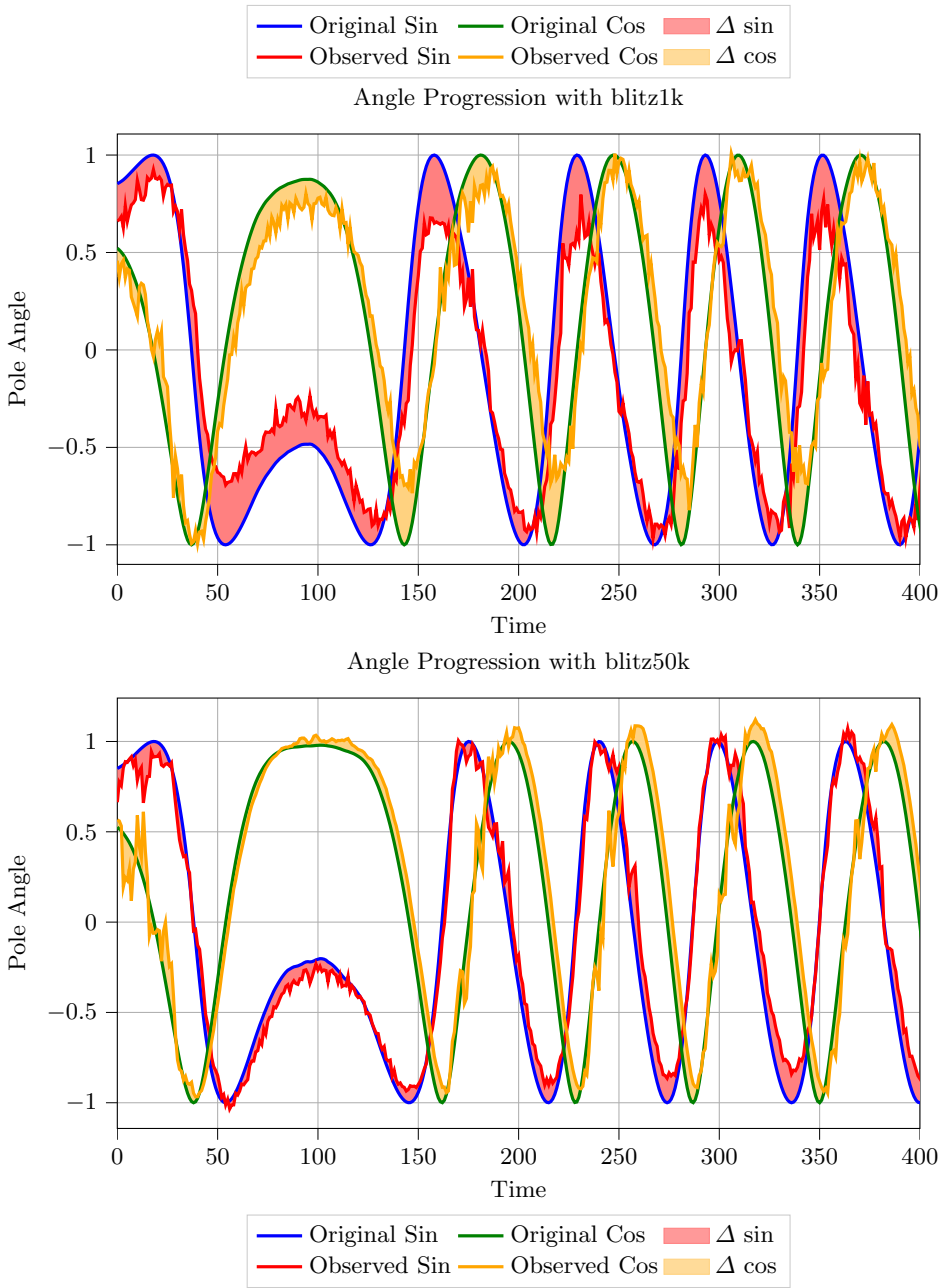

Fig. 11: This figure shows how the NN performed with 1k training time series and 50k time series. See Figure 13 the appendix for 5k and 500k plots.

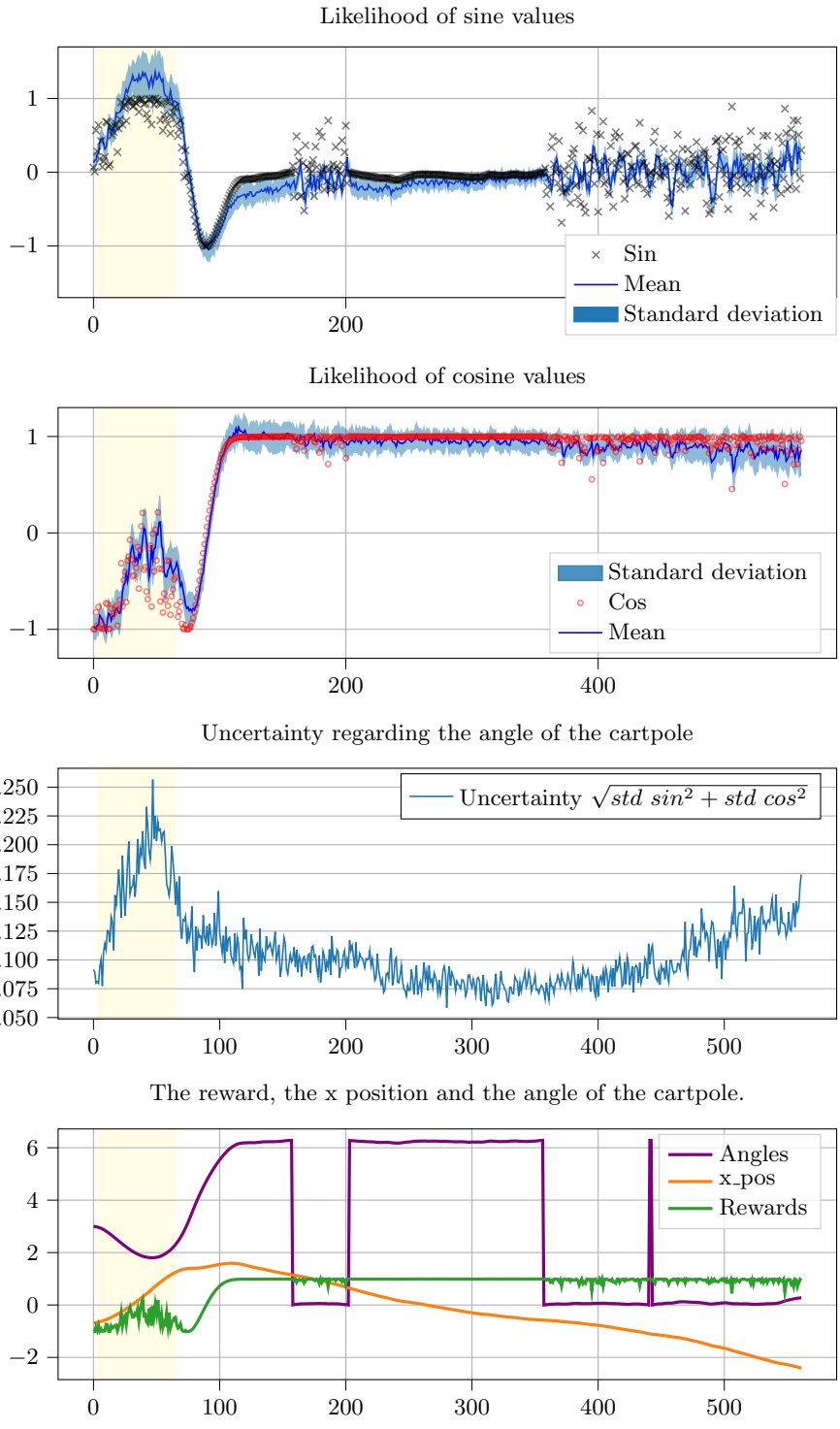

Fig. 12: Different values over 550 time steps.

Figure 12 shows the predictions of the neural network together with the mean and the standard deviation. The uncertainty of the network is smaller in the yellow area. This is the right part of the CartPole circle, which had no noise added. Outside this area, the network is not so sure about its predictions, therefore the standard deviation is bigger.

The next step was to use the PPO algorithm offered by stable_baselines to train an agent on our environment. To make use of the uncertainty offered by our environment, we had to create new reward functions to make use of this information.

5.2 Reward Functions

Our implementation allows to experiment with different reward functions in a plug and play fashion. The user can pass any reward function as a `Callable` to the environment and set up their own trials.

As the ultimate goal of the agent is to swing up the pendulum, we need to incentivize this behavior. The environment uses the unit circle, but turned 90° anti-clockwise. If we recall Figure 9, this means whenever the pendulum is upright, $cos(\alpha) = 1$ and < 1 otherwise. Thus, it seems logical to uses the cosine value as a base reward function:

```
reward = cosine
```

Obviously, this function does not consider the uncertainty. So we multiply the cosine value with 1 minus the uncertainty. So high uncertainty discounts the reward, while low uncertainty is rewarded:

```
reward = cosine * (1 - uncertainty)
```

Since we want our agent to learn the environment, but also avoid areas with a higher uncertainty, we expect this function to perform rather well.

During training, we often observed, that our cartpole drove out of bounds, thus ending the run of the agent. To refrain the cart from doing that while also improving learning time, we thought that it might be a good idea, to also incorporate the x position of the cart. So we subtract the modulus of it multiplied by a scaling factor:

```
reward = cosine * (1 - uncertainty) - abs(x_pos * x_pos_weight)
```

By subtracting the weakened `x_pos` from our reward, we expect the agent to learn the environment without moving too far from the center.

However, we found that by subtracting a weakened uncertainty from cosine and using this as reward, we were able to achieve the best results. We combined the uncertainty of the angle parts into a single uncertainty.

```
reward = cosine - (math.sqrt(std_sin ** 2 + std_cos ** 2)*0.1)
```

5.3 Algorithms

Stable Baselines3 offers a comprehensive set of RL algorithms (see Table 1 below). We used PPO because it is easy to use and reaches good performances. It is comparable or superior to state-of-the-art algorithms, while being easier to tune and easier to implement.

Table 1: Overview of the implemented algorithms in Stable Baselines3 [20]. *Box* is an N-dimensional box that contains every point in the action space. *Discrete* refers to a list of possible actions, where each time step only one of the actions can be performed. *MultiDiscrete* on the other hand is a list of possible actions, where each time step only one action of each discrete set can be used. Finally, *MultiBinary* refers to a list of possible actions, where each time step any of the actions can be used in any combination.

Name	Box	Discrete	MultiDiscrete	MultiBinary
A2C	✓	✓	✓	✓
DDPG	✓	✗	✗	✗
DQN	✗	✓	✗	✗
HER	✓	✓	✗	✗
PPO	✓	✓	✓	✓
SAC	✓	✗	✗	✗
TD3	✓	✗	✗	✗

6 Evaluation

6.1 Results

Overall, we could observe, that the trained PPO agent learned to try to avoid the noisy spaces. At every start of a run, the agent tries to spin the pole to the top along the right side, because the left side was the noisy space in our environment. Due to this noise, it is very hard to display the result in a plot. That is why we recommend, that the user clones our repository and runs the main.py class without any modifications. Another observation we made, was that PPO was not able to properly solve our environment, but neither was it able to solve the original environment. Future investigations will show, if hyperparameter-tuning or the usage of another RL algorithm could lead to more favorable results.

6.2 Limitations

The nature of machine learning and artificial intelligence depends to a high degree on nondeterminism. The methods and frameworks used are partly non-deterministic, so in order to say that two things are actually different, we have to deploy statistical testing. Since we did not do this our experiments are inherently biased.

6.3 CancelOut

CancelOut [6] is a deep neural network layer that can assist in identifying a subset of important input characteristics for streaming or static data. The core idea is to update `CancelOut` weights (W) during a training stage, such that *noisy* or less important characteristics are canceled out with a negative weight. Otherwise, the best characteristics that contribute the most to the learning process will be passed on. However, extensive tests revealed that this kind of layer smoothed the predictions too much, so we removed it from the final release.

7 Conclusion

With the abundance of approaches and different algorithms, it can be hard to keep track of how useful these techniques are in concrete problems. In this paper we showed that uncertainty in machine learning remains an uncertainty in itself, how default parameters often need to be adjusted for the specific problem, and that learning or teaching, respectively, remains an inherent difficult and complex task to solve for machines. While the right settings are able to reduce the reward variance, agents need a lot of time and computing power to achieve satisfactory results consistently. Going forward, the training set should cover a larger sample set, and it may be of interest to investigate if a time series as an input to the neural network is really necessary. As we recall from the beginning of this paper, the Markov property states that the best prediction for an event does not change with the information state. Whether one knows the whole process up to s or only the current state, the prediction for the further course of the process is not changed. On a similar note, it might also be worth simulating different parameters like the position of the cart or the outage of sensors. Another interesting aspect is the performance of other RL algorithms compared to PPO. Future research should employ larger computational budgets and more iterations to evaluate the maximum capabilities of these algorithms with regard to uncertainty.

References

1. Tikz: Annotate edges with distributions of bayesian neural network - tex - latex stack exchange, https://tex.stackexchange.com/questions/503590/tikz-annotate-edges-with-distributions-of-bayesian-neural-network/503646#503646
2. Achiam, J.: Spinning Up in Deep Reinforcement Learning (2018), https://spinningup.openai.com/
3. Barto, A.G., Sutton, R.S., Anderson, C.W.: Neuronlike adaptive elements that can solve difficult learning control problems. IEEE Transactions on Systems, Man, and Cybernetics **SMC-13**(5), 834–846 (1983). https://doi.org/10.1109/TSMC.1983.6313077
4. Benmiloud, M.: Free body diagram of an inverted pendulum in tikz - tikzblog, https://latexdraw.com/free-body-diagram-of-an-inverted-pendulum-in-tikz/
5. Blundell, C., Cornebise, J., Kavukcuoglu, K., Wierstra, D.: Weight uncertainty in neural networks (2015)
6. Borisov, V.: unnir/cancelout: Cancelout is a layer for deep neural networks, that can help identify a subset of relevant input features for streaming or static data., https://github.com/unnir/CancelOut
7. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym (2016), https://gym.openai.com/
8. Depeweg, S.: Modeling Epistemic and Aleatoric Uncertainty with Bayesian Neural Networks and Latent Variables. Ph.D. thesis, Technical University of Munich, Germany (2019), https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20191001-1482483-1-8
9. Esposito, P.: Blitz - bayesian layers in torch zoo (a bayesian deep learing library for torch). https://github.com/piEsposito/blitz-bayesian-deep-learning/ (2020)
10. Hüllermeier, E., Waegeman, W.: Aleatoric and epistemic uncertainty in machine learning: an introduction to concepts and methods. Machine Learning **110**(3), 457–506 (Mar 2021). https://doi.org/10.1007/s10994-021-05946-3, https://doi.org/10.1007/s10994-021-05946-3
11. Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., Bridgland, A., Meyer, C., Kohl, S.A.A., Ballard, A.J., Cowie, A., Romera-Paredes, B., Nikolov, S., Jain, R., Adler, J., Back, T., Petersen, S., Reiman, D., Clancy, E., Zielinski, M., Steinegger, M., Pacholska, M., Berghammer, T., Bodenstein, S., Silver, D., Vinyals, O., Senior, A.W., Kavukcuoglu, K., Kohli, P., Hassabis, D.: Highly accurate protein structure prediction with AlphaFold. Nature **596**(7873), 583–589 (Jul 2021). https://doi.org/10.1038/s41586-021-03819-2, https://doi.org/10.1038/s41586-021-03819-2
12. Kamperis, S.: Probabilistic regression with tensorflow — a blog on science, https://ekamperi.github.io/machine%20learning/2021/01/07/probabilistic-regression-with-tensorflow.html
13. Kiureghian, A.D., Ditlevsen, O.: Aleatory or epistemic? does it matter? Structural Safety **31**(2), 105–112 (2009). https://doi.org/10.1016/j.strusafe.2008.06.020, https://www.sciencedirect.com/science/article/pii/S0167473008000556, risk Acceptance and Risk Communication
14. Ângelo Lovatto: angelolovatto/gym-cartpole-swingup: A simple, continuous-control environment for openai gym, https://github.com/angelolovatto/gym-cartpole-swingup

15. MacKay, D.J.C.: Information-Based Objective Functions for Active Data Selection. Neural Computation **4**(4), 590–604 (07 1992). https://doi.org/10.1162/neco.1992.4.4.590, https://doi.org/10.1162/neco.1992.4.4.590
16. Mahmud, M., Kaiser, M.S., Hussain, A., Vassanelli, S.: Applications of deep learning and reinforcement learning to biological data. IEEE Transactions on Neural Networks and Learning Systems **29**(6), 2063–2079 (2018). https://doi.org/10.1109/TNNLS.2018.2790388
17. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (Feb 2015). https://doi.org/10.1038/nature14236, https://doi.org/10.1038/nature14236
18. Moberg, J.: Uncertainty-aware models for deep reinforcement learning. Master thesis, Chalmers tekniska högskola / Institutionen för elektroteknik (2019), https://hdl.handle.net/20.500.12380/300314
19. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
20. Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., Dormann, N.: Stable baselines3. https://github.com/DLR-RM/stable-baselines3 (2019)
21. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. CoRR **abs/1707.06347** (2017), http://arxiv.org/abs/1707.06347
22. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (Jan 2016). https://doi.org/10.1038/nature16961
23. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D.: A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. Science **362**(6419), 1140–1144 (Dec 2018). https://doi.org/10.1126/science.aar6404, https://doi.org/10.1126/science.aar6404
24. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., Hassabis, D.: Mastering the game of go without human knowledge. Nature **550**(7676), 354–359 (Oct 2017). https://doi.org/10.1038/nature24270, https://doi.org/10.1038/nature24270
25. Sountsov, P., Suter, C., Burnim, J., Dillon, J.V.: Regression with probabilistic layers in tensorflow probability — the tensorflow blog, https://blog.tensorflow.org/2019/03/regression-with-probabilistic-layers-in.html

26. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning. MIT Press, Cambridge, MA, USA, 1st edn. (1998), http://incompleteideas.net/book/the-book.html
27. Taskjelle, T.: https://tex.stackexchange.com/questions/152054/unit-circle-sinx-radians-on-x-axe/152056#152056
28. Todorov, E., Erez, T., Tassa, Y.: Mujoco: A physics engine for model-based control. In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 5026–5033 (2012). https://doi.org/10.1109/IROS.2012.6386109
29. Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J.P., Jaderberg, M., Vezhnevets, A.S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T.L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., Silver, D.: Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature **575**(7782), 350–354 (Oct 2019). https://doi.org/10.1038/s41586-019-1724-z, https://doi.org/10.1038/s41586-019-1724-z
30. Watkins, C.J.C.H.: Learning from Delayed Rewards. Ph.D. thesis, King's College, Cambridge, UK (May 1989), http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf

A Additional Graphics

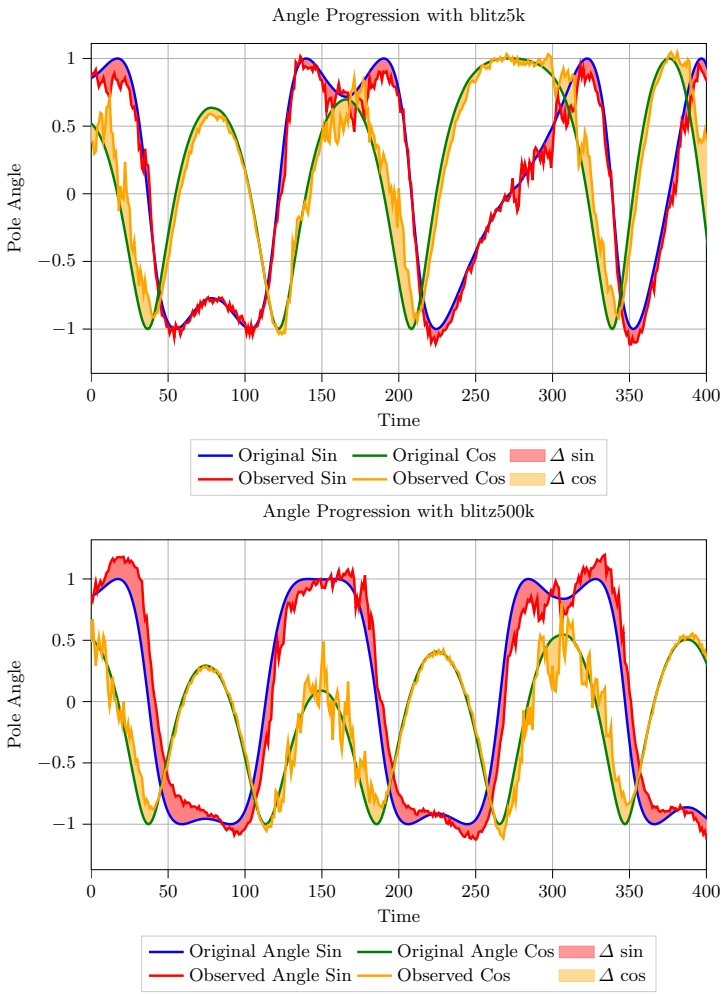

Fig. 13: This figure shows how the NN performed with 5k training time series and 500k time series.