Technische Universität Berlin

Hypernetworks - Neural Networks that Generate Neural Networks

by

Georg Kruse

Matriculation Number 451138

A thesis submitted to

Technische Universität Berlin
School IV - Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Neural Information Processing

Master's Thesis

30.06.2021

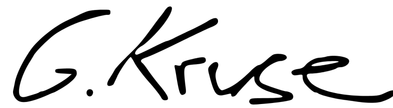
Supervised by:
Prof. Dr. Klaus Obermayer

Assistant supervisor:
Dr. Vaios Laschos

Statutory Declaration

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

The independent and unaided completion of the thesis is affirmed by affidavit:



Berlin, June 21, 2021 Georg Kruse

Abstract

In reinforcement learning (RL) gradient based methods have shown to efficiently master many games in a superhuman manner. New advances in meta learning aim to improve these playing agents in terms of sample efficiency by introducing new methods of 'learning how to learn'. They also try to enhance the adaptation capabilities of the agents to quickly changing and diverse playing styles of their opponents. To study and evaluate these approaches, a diverse set of agents is needed.

In this work hypernetworks will be evaluated as a method to create a diverse set of agents. Hypernetworks are neural networks that generate weights for another neural network. They can be used in a similar fashion as variational autoencoders (VAEs) or generative adversarial networks (GANs) to generate a mapping from a low dimensional (random) input vector to high dimensional weight space. The loss function of such a hypernetwork contains two properties: accuracy and diversity. The latter is used to ensure that the network generates a distribution of agents.

The approach is bench marked against individually trained agents as well as an evolutionary method, where the selection criterion consists also of both, an accuracy and a diversity term. The different approaches are tested on an image classification task, a graph game and an imperfect information card game.

Contents

1 Introduction **1**

2 State of the Art **3**
2.1 Reinforcement Learning . . . 3
2.1.1 Double Deep-Q-Network . . . 4
2.1.2 Actor-Critic . . . 5
2.1.2.1 Proximal Policy Optimization . . . 7
2.2 Hypernetworks . . . 7
2.3 Evolutionary Methods . . . 11
2.4 Diversity . . . 12

3 Methods and Concept **15**
3.1 Hypernetworks for Reinforcement Learning . . . 15
3.1.1 Loss Function . . . 16
3.1.1.1 Cosine Similarity . . . 16
3.1.1.2 Diversity via Determinants . . . 17
3.1.2 Architecture . . . 17
3.2 Evolutionary Strategy . . . 19

4 Experiments **21**
4.1 Image Classification . . . 21
4.1.1 Accuracy . . . 21
4.1.2 Diversity . . . 22
4.2 Graph Game . . . 23
4.3 Imperfect Information Game . . . 27

5 Conclusion **33**

List of Tables **35**

List of Figures **35**

Bibliography **37**

Appendices **41**

Appendix 1 **43**

Appendix 2 **45**

1 Introduction

In his talk *'Hypernetworks: a versatile and powerful tool'*[1] Prof. Wolf from the Tel Aviv University and Facebook AI Research stated that he thinks that the world is at the edge of a second Cambrian explosion. This explosion refers to a process over 500 mio. years ago in which a lot of the evolutionary steps, which shape our life nowadays, happened in a rather short period of time. He believes that this is exactly what is going to happen in the next decades with AI.

One key aspect which has been boosting his research and might have led him to this statement was the usage of hypernetworks: neural networks which generate the weights of other neural networks. The name originates from a paper published in 2016 and since then many researchers have used this approach in their research.

Another field with exploding novelty is reinforcement learning (RL). Media-effective events like the Go tournament against the world champion Lee Sedol 2017 and the Dota2 tournament against the world champion Team OG 2019 express the ongoing progress pushed by a growing research community. Major players like Deepmind, OpenAI and Facebook AI together with universities and other research organizations reach milestone after milestone.

Most of the basic ideas in RL like the Bellman equation have already been around for decades. The breakthrough of these ideas in recent years is on one hand due to the huge increase in computational power. On the other hand, main advances were possible because of the combination of several existing ideas: The new combination of policy networks, value networks and monte carlo tree search (MCTS) for example, led to the incredible success of AlphaGo [1] and MuZero [2].

Another new combination of existing ideas is hypernetworks and RL. This combination seems promising in the field of agent diversification. Hypernetworks could help finding agents who not only reach optimal scores in the given task, but also find diverse ways to do so. Especially in tasks which require severe exploration [3] or require cooperative behavior between multiple agents [4], exploring a diverse agent-space facilitates completion.

Furthermore, diverse sets of agents are also needed to evaluate recent advances in meta learning. Meta learning, which can be thought of as a 'learning how to learn', tries to improve the sample efficiency as well as the the adaptation capabilities of current RL agents. In order to develop agents capable of adapting quickly to changing playing styles with meta learning approaches, a diverse set of agents is needed.

In order to explore this idea of combining hypernetworks with state of the art RL algorithms to create diverse agents, several steps are conducted in this thesis. First, the overall capability of a hypernetwork to establish diversity is analyzed by reproducing the work of Deutsch. In his paper *'Generating neural networks with neural networks'*[5] he showed that hypernetworks are able to create diverse classifier networks for image classification. Second, this approach is

[1] Find his talk at https://www.youtube.com/watch?v=KY9DoutzH6k

transferred to RL. Criteria to measure and estimate diversity are explored and combined with the created hypernetwork to solve a simple RL-task, namely a graph game. Third, the learned insights of this experiment are used to solve a difficult RL-task: an imperfect information game. In order to compare the diversity of the agents created by the hypernetwork they are benchmarked against agents which are either individually trained with random weight initializations or created by a evolutionary method, a genetic algorithm.

This thesis is therefore structured in four parts. In *Chapter* 2 the basic ideas utilized are presented to give an overview of the research landscape in the respective fields. In *Chapter* 3 the main ideas and concepts of the thesis are presented. The experiments to evaluate and analyze these ideas are conducted in *Chapter* 4. Lastly, the main insights and findings are summarized in *Chapter* 5 and suggestions for future research are outlined.

2 State of the Art

The following chapter is divided into four parts, providing basic information and examples about the main concepts utilized in this thesis: *reinforcement learning*, *hypernetworks*, *genetic algorithms* and *diversity measurements*.

2.1 Reinforcement Learning

In RL a so called agent interacts with an environment and receives some sort of reward depending on the taken actions. Many different algorithms have been introduced for such agents. An overview by OpenAI [6] can be seen in 2.1.[1]

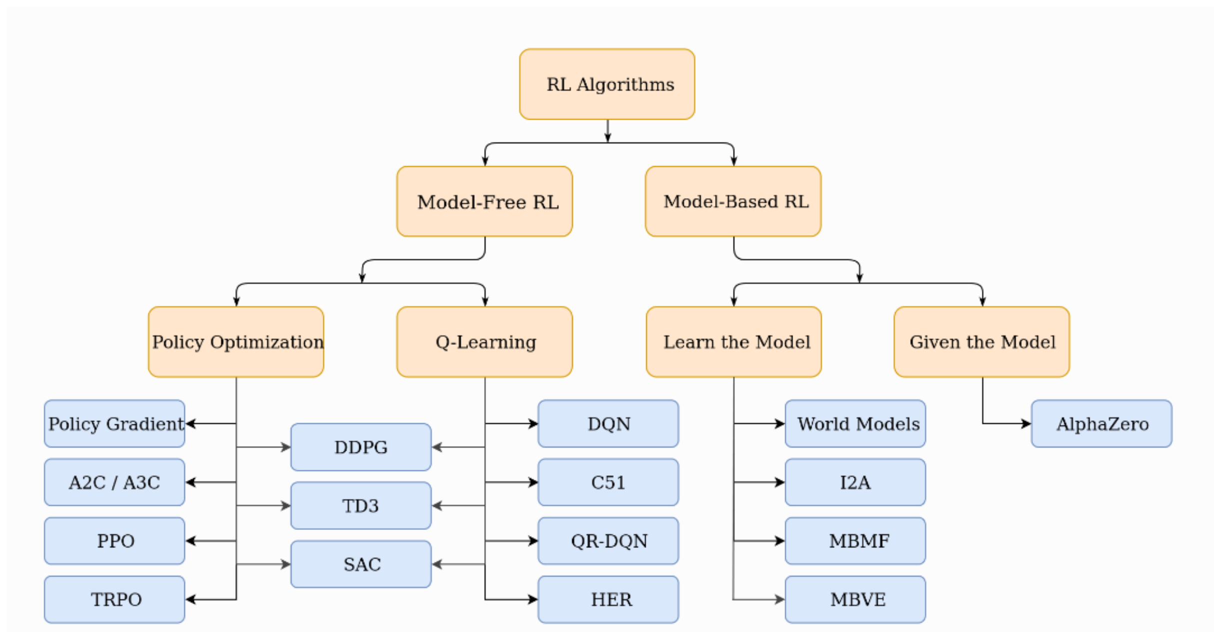


Figure 2.1: A taxonomy of algorithms in RL: most algorithms can be categorized by this tree into model-free RL, which has been more popular among researchers, and model-based RL

In this thesis two model-free algorithms are implemented. One from the field of *Policy Optimization*, an advantage actor-critic (A2C), and one from *Q-Learning*, a Double Deep-Q-Network (DDQN). These RL algorithms will interact with an environment in a discrete Markov decision process (MDP) with a finite number of states and actions and bounded rewards.

The training of an agent generally starts with an initial state s_0. A trajectory $\tau = (s_0, a_0, s_1, a_1, ...)$

[1] Find his figure at https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

is created by picking actions a_t at time step t according to a policy $\pi_\theta(a_t|s_t)$ and receiving states by the environment according to the transition model $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$ until a terminal state is reached [7]. For these actions and states a reward r_t is received according to a reward function $r(s_t, a_t, s_{t+1})$. The goal of the agent is to maximize this cumulative reward $R(\boldsymbol{\tau}) = \sum_{t=0}^{T} r_t$ [8].

The form of the objective function to maximize the cumulative reward, differs between *Q-Learning* and *Policy Optimization* algorithms. In the following the characteristics of the two RL algorithms used in the thesis are introduced.

2.1.1 Double Deep-Q-Network

In Q-Learning, the objective function is an approximator $Q(s, a)$ of the optimal state-action value function $Q^*(s, a)$ which is approximated via stochastic gradient descent [9]. As has been shown above, the best action under a given policy is the one that maximizes the expected return. This is called a greedy policy. In equation 2.1 it is shown that this function can be estimated by the reward of the current time step, plus an estimation of the value of the next time step, given a state s_t and action a_t. γ is a temporal discount factor.

$$Q^*(s_t, a_t) = \mathbb{E}_{s_{t+1} \sim P}\left[r(s_t, a_t) + \gamma \max_{a_{t+1}}[Q^*(s_{t+1}, a_{t+1})\middle|s_t, a_t\right] \tag{2.1}$$

The optimal policy is derived by simply selecting the highest value-action pair for each state. The DDQN is a neural network which approximates these value-action pairs. It maps the state space to the action space by training its parameters θ. To do so efficiently, two important ingredients are added: a target network, which has the same architecture as the so called online network, but has a different set of parameters θ_T and a replay buffer.

The problem of the standard DQN approach is that the same parameters are used to select and to evaluate the action. This leads to the overestimation of the Q-values and to inaccuracies during training [7]. To fix this issue, the DDQN approach estimates the value of the greedy policy with the online network with weights θ.The target network calculates the value estimation with a second value function with a different set of weights θ_T. Therefore, the target becomes

$$y^{DDQN} = r(s_t, a_t) + \gamma Q_{\theta_T}\left(s_{t+1}, \arg\max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1})\right). \tag{2.2}$$

Note that equation 2.2 is considered one-step Q-learning, since it only takes the following time step $t + 1$ into account. The estimated value of an action is determined by the reward at the next time step and the estimated value of the best action times a discount factor γ, which takes the temporal differences into account. Note that the value is calculated by the target network, whereas the online action is selected by the online network. The loss function for the DDQN then becomes

$$\mathcal{L}^{DDQN}(\theta) = \mathbb{E}\left(y^{DDQN} - Q_\theta(s_t, a_t)\right)^2. \tag{2.3}$$

The weights of the target network θ_t are equated every n update steps to the weights of the online network. n has to be chosen carefully since it highly influences the stability of the DDQN. The same holds for the parameter γ.

Another method to boost performance is experience replay, where the transitions are simply saved in memory and from there randomly sampled for training. The training therefore

happens in an offline manner (off-policy).

In the beginning of training, exploration is important for the success of a RL algorithm. Since the DDQN always selects the action with the maximal Q-value, a strategy has to be introduced to ensure that, while the Q-values estimates are imprecise, the DDQN also explores new state-action pairs. This can be achieved by a (linearly or exponentially) decreasing parameter $\epsilon \in 0, 1$. Instead of always taking the action with the highest Q-value, in the beginning of training the agents take a random action with probability ϵ. This ensures exploration and is called an ϵ-greedy policy.

The pseudo code of the DDQN used in this thesis can be seen below (**Algorithm 1**).

Algorithm 1 Double DQN [7]

```
Initialize online network Q_θ, target network Q_θT and replay buffer D
for each iteration do
  for each step in enviroment do
    take s_t as input and select a_t ~ π_θ(a_t, s_t)
    execute a_t and observe next state s_{t+1} and reward r_t
    store trajectory {s_t, a_t, r_t, s_{t+1}} → D
  end for
  for each update step do
    sample τ_t = (s_t, a_t, r_t, s_{t+1}) ~ D
    Compute target: y^DDQN = r_t + γ Q_θT(s_{t+1}, argmax_{a_{t+1}} Q_θ(s_{t+1}, a_{t+1}))
    Perform gradient descent step on loss MSE(y^DDQN, Q_θ(s_t, a_t))
  end for
  Update target network parameters θ_T = θ every i'th iteration
end for
```

2.1.2 Actor-Critic

In policy optimization, the objective function maximizing the cumulative reward is

$$J(\pi_\theta) = \mathbb{E}_{\boldsymbol{\tau}\sim\pi_\theta}\Big[R(\boldsymbol{\tau})\Big]. \tag{2.4}$$

where π_θ is the policy of the agent with parameters θ and $\boldsymbol{\tau}$ are the transitions due to this policy. The gradient can be approximated by sampling actions from $\pi_\theta(a_t|s_t)$ [9, 10]. The gradient of the objective function can be rewritten as

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\Big[\sum_{t=0}^{T} \nabla_\theta \log\pi_\theta(a_t|s_t) \cdot A_t\Big]. \tag{2.5}$$

As can be seen in equation 2.5, the A2C consist of two parts: an advantage function A_t and the policy function $\pi_{\theta_a}(a|s)$. This is due to the structure of the overall algorithm. The value-network $V^{\pi}_{\theta_c}(s)$ called the critic approximates the advantage function and a policy-network called the actor approximates the policy function $\pi_{\theta_a}(a|s)$ [10, 11]. θ_a and θ_c are the weights of the networks. Note that these weights can be partially shared between the networks, but this will not be considered in the following notation.

For a given time step t the action a_t is selected according to the policy network. For a given

state it outputs a probability distribution over the action space and selects an action a_t by sampling from this distribution.

The A2C is trained directly on the data from the played episode. It's an on-policy method and therefore, no replay buffer is needed. At each time step [t], the value of the state is estimated by the critic . The value network estimates the reward received at the time step. The loss of the critic can be written as

$$\mathcal{L}_{\text{critic}} = \mathbb{E}_{s_t \sim \pi_{\theta_a}} \left(V^{\pi}_{\theta_c}(s_t) - r_t \right)^2 \tag{2.6}$$

The loss of the actor is estimated by combining the log policy with the advantage function. The advantage of a state is the received reward plus the expected reward at the next time step minus the reward at the current time step. The expected reward of the next time step is multiplied by a discount factor γ to prioritize immediate rewards (see equation 2.7).

$$A_t = r_t + \gamma V^{\pi}_{\theta_c}(s_{t+1}) - V^{\pi}_{\theta_c}(s_t) \tag{2.7}$$

Equivalent to equation 2.5 the loss of the actor is

$$\mathcal{L}_{\text{actor}} = \mathbb{E}_{s_t, a_t \sim \pi_{\theta_a}} \left(\log \pi_{\theta_a}(a_t|s_t) \cdot A_t \right) + \delta \mathcal{L}_{\text{entropy}}. \tag{2.8}$$

To account for a better exploration, an entropy term $\mathcal{L}_{\text{entropy}}$is added to the loss of the policy network. The parameter $\delta \in \{0, 1\}$ is the entropy coefficient and decrease over the course of training. The overall loss of the A2C is then simply the sum of the losses of the policy and value network (see equation 2.9). The critic loss is multiplied by a factor $a < 1$ and is usually chosen to be 0.5. The A2C can then be trained with this combined loss via gradient ascent.

$$\mathcal{L}_{\text{A2C}} = \mathcal{L}_{\text{actor}} + a \cdot \mathcal{L}_{\text{critic}} \tag{2.9}$$

Algorithm 2 Actor Critic (PPO) [12]

Initialize policy parameters θ_a and θ_c
for for each iteration **do**
  for each enviroment step **do**
    collect set of trajectories $\mathcal{D} = \{\tau_i\}$ by selecting $a_t \sim \pi_\theta a(a_t, s_t)$
    $\rightarrow \{s_t, a_t, r_t, s_{t+1}\}$
  end for
  for e in range epochs **do**
    sample mini batch from $\mathcal{D}$ and calculate:
    $A_t = r_t + \gamma V^{\pi}_{\theta_c}(s_{t+1}) - V^{\pi}_{\theta_c}(s_t)$
    $p_t = \pi_{\theta_a}(s_t)$
    $\mathcal{L}_{\text{critic}} = (V^{\pi}_{\theta_c}(s_t) - r_t)^2$
    $\mathcal{L}_{\text{policy}} = -\log(p_t(a_t))A_t$
    $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{policy}} + \mathcal{L}_{\text{value}} + \mathcal{L}_{\text{entropy}}$
    Update policy and value network with gradient ascent
  end for
end for

2.1.2.1 Proximal Policy Optimization

Because policy gradient methods tend to be unstable compared to Q-learning methods, proximal policy optimization (PPO) was introduced [13]. The main idea behind PPO is gradient clipping (see figure 2.2). The approach tries to prevent the size of gradient updates, which diverge from the previous gradient updates. This is supposed to make the learning more stable.

The main objective proposed is the following:

$$\mathcal{L}^{\mathrm{CLIP}}(\theta) = \mathbb{E}_t \min(r_t(\theta)A_t, \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t) \tag{2.10}$$

where epsilon is a hyperparameter, normally $\epsilon = 0.2$ [13]. Instead of multiplying the log of the policy $\pi_\theta(a_t|s_t)$ with the advantage function, the ratio between the current policy and the old policy is used (see equation 2.11).

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)} \tag{2.11}$$

The min-function indicates that either this ratio or a clipped value is used to estimate the loss. A graphical interpretation of this approach can be seen in figure 2.2. The gradient is clipped off for both positive and negative values of the advantage function, to prevent large gradient steps in diverging directions.

The pseudo code of the A2C used in this thesis can be found in **Algorithm 2**.

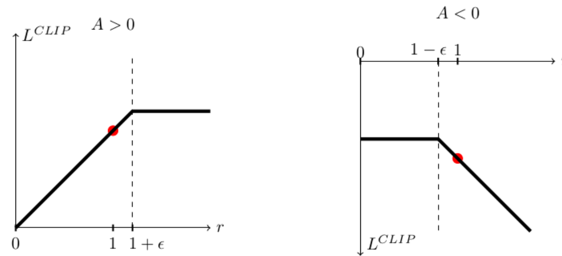


Figure 2.2: The plots show the surrogate function of $\mathcal{L}^{\mathrm{CLIP}}$ as a function of the ratio r for positive advantages (left) and negative advantages (right). The red circle indicates the starting point of the optimization.[13]

2.2 Hypernetworks

The name hypernetwork originates from the eponymous paper '*Hypernetworks*' by Ha et al. and initially described an approach using a smaller network to generate the weights of a larger network [14, 15]. The weight generating network is referred to as hypernetwork, whereas the other network is called main network.

Since then, many different applications of such hypernetworks have been introduced. Therefore, the initial typification in small and large network is obsolete because hypernetworks now come in almost any size and shape. Without claiming generality and albeit different classifications are possible, three main trends can be identified.

1. Hypernetworks can take additional information about the task as inputs and modify the weights of the main network directly instead of transforming its output. By doing so they are able to solve problems more efficiently. The hypernetwork serves in this case as an auxiliary network. Since this approach will not be used in this thesis but still offers some insights into the overall functioning of hypernetworks, some examples will be briefly described below.

2. Instead of taking additional information about the task as inputs, hypernetworks can also use a noise distribution as input to generate the weights of a main network. This is a generative method similar to VAE's and GAN's and will be elaborated below.

3. Hypernetworks can be used as a compression method for larger main networks. Instead of having to save all parameters of a main network, a much shorter vector representation can be used which can, when fed into a prior trained hypernetwork, reproduce the entire parameter set of the main network. Since this approach can be considered to be a different field of research, it won't be treated in this thesis.

1. Examples: Using additional information

Meta Learning: In their paper *Continual Learning with Hypernetworks*, von Oswald et al. used hypernetworks to tackle problems in the field of continual learning. If several tasks t are given $\{(\mathrm{x}^{(1)}, \mathrm{y}^{(1)}), ..., (\mathrm{x}^{(t)}, \mathrm{y}^{(t)})\}$ with input samples $\mathrm{x}^{(t)}$ and output samples $\mathrm{y}^{(t)}$, a standard approach to train the model is to use data from all tasks. In real world application this isn't always possible due to catastrophic forgetting. An alternative approach is using a task-conditioned hypernetwork. Such a hypernetwork h doesn't have to retrain the information of the previous tasks within its set of parameters, because it can map a task specific embedding $\mathrm{e_t}$ to a task specific set of weights $\mathrm{h}(\mathrm{e_t}) \rightarrow \theta_\mathrm{t}$ of a main network. The embeddings are used to memorize the tasks. Note that the task embedding $\mathrm{e_t}$ is a differentiable deterministic parameter and therefore can be learned just like the weights θ_t by minimizing the task specific loss $\mathcal{L}^{(\mathrm{t})}_{\mathrm{task}}$. One key challenge remaining is to determine which task shall be solved from a given input pattern, if the task identity and therefore the embedding e is not ambiguous [16].

Meta Pruning: Another approach using network encoding vectors as inputs for a hypernetwork is proposed by Zechun Liu et al. [17]. They used the hypernetwork for channel pruning (pruning is an efficient method to compress and accelerate neural networks). Doing pruning manually was shown to be non-trivial, since pruning one channel in one layer might influence the following layers significantly. The so-called *Meta Pruning* approach proved to outperform uniform pruning baselines as well as other state-of-the-art pruning methods. Instead of modifying the main networks manually, hypernetworks could solve the task more efficiently by generating the pruned main networks di-

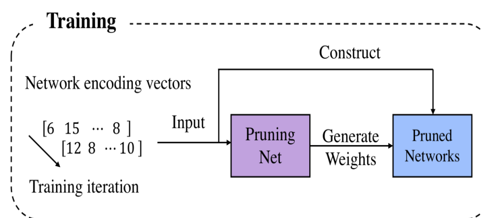


Figure 2.3: The pruning net (hypernetwork) takes encoding vectors of the main networks as input to generate the weights of the pruned main networks.[17]

rectly. As can be seen in figure 2.3, the encoding vectors of the main network were used to generate a pruned version of the networks.

One-Shot-Learners: Luca Bertinetto et al. used a hypernetwork-main network approach (albeit calling it a learnet and a pupil network) for one-shot learning [18]. To create their learners they reformulated the training problem. The task was a simple image classification task, but instead of simply predicting the labels of a given picture with the conventional approach (which isn't sample efficient), they created input-triplets (x_i, z_i, l_i) where x_i was the picture to be predicted, z_i an exemplar of a class of interest of such pictures and l_i a modified label, which would turn to one, if x_i and z_i belong to the same class and negative otherwise. They then fed x_i to a pupil network (main network) and z_i to the learnet (hypernetwork). As can be seen in figure 2.4, the hypernetwork, which takes the class representative z_i as input, manipulates the weights of the second layer of the main network as well as the output of the network. The overall output Γ is the prediction of the label l_i of the image x_i. This approach managed to outperform other siamese architectures in the one-shot learning scenario.

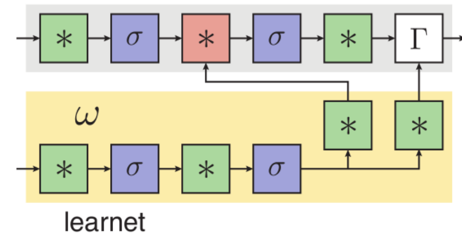


Figure 2.4: The above networks are the pupil network (above) and learnet (below). The stars represent the layers of the networks and σ the activation functions between them. Γ represents the overall output. [18]

Meta Functionals: Wolf et al. show that hypernetworks can be used to solve difficult tasks by reformulating the initial problem. The given problem in the paper '*Deep Meta Functionals for Shape Representation*' is to reconstruct a 3D model from a single image, for example a plane as can be seen in figure 2.5. This task was reformulated in a way that a hypernetwork could be used to solve it. First, the hypernetwork f receives the image as input and outputs the weights of the main network (see 2.6). This main network is therefore a function of the image. It serves as a classifier which maps randomly sampled points with coordinates (x,y,z) to values $\{0, 1\}$: 1 if the point is within the 3D shape of the image and 0 if its outside. The 3D shape of the image is thereby defined by the classifiers decision boundary (see figure 2.5). Wolf et al. also showed in other publications that a similar approach can be pursued for a variety of difficult tasks such as electric circuit modeling [19] or color enhancement [20],

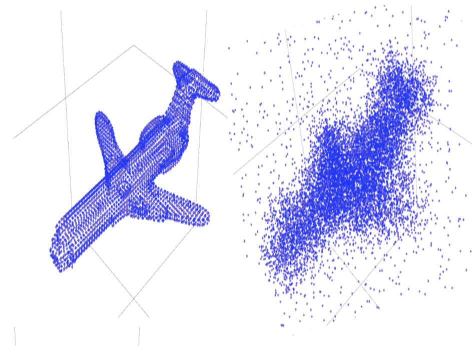


Figure 2.5: The sampling of the points which are classified by the network g can be seen (right) as well as the target shape of an image, e.g. a plane (left)

which can be solved by converting the initial problem into a hypernetwork problem.

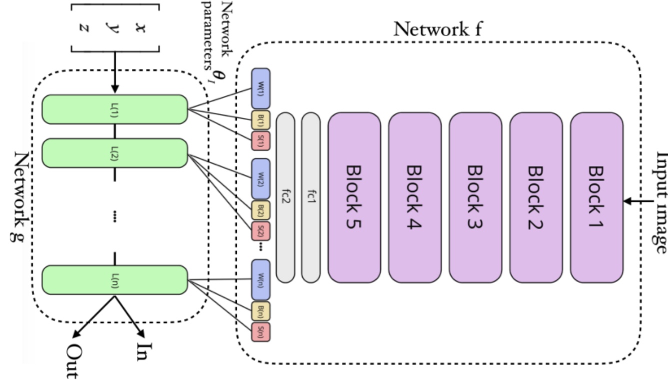


Figure 2.6: The hypernetwrok f takes an image as input and outputs the weights of the classifier network g. This network takes coordinates as input and determines if they are inside or outside the 3D-shape of the image.[21]

Hyperparameter Tuning: As a last use case it shall be mentioned that Jonathan Lorraine et al. showed that instead of using cross-validation for hyperparameter tuning, hypernetworks can be used to find the optimal hyperparameters of a main network [22]. Especially in the case of tuning thousands of hyperparameters this approach proved to be efficient.

2. Examples: Generative methods

Generating Neural Networks with random input vectors: Because VAEs [23] and GANs [24] have shown impressive results in generating samples from complex and high-dimensional distributions, Deutsch proposed extending this idea to neural networks [5]. Considering that there are no datasets of neural networks for each task, hypernetworks cannot be trained like the before mentioned generative methods. An underlying probability distribution of neural networks does not exist. Thus, useful properties of such a distribution have to be chosen as a selection criteria. Deutsch selected **accuracy** and **diversity** as criteria.The training objective of the introduced hypernetwork is a compromise between these two criteria.

If $\mathcal{M}(\mathrm{x};\theta) : \mathrm{X} \times \theta \rightarrow \mathrm{Y}$ is the main network, where X is the input and Y the output of the network and θ the set of trainable weight vectors of the network, then the hypernetwork or generater network can be written as $\mathcal{G}(\mathrm{z};\varphi) : \mathrm{Z} \times \varphi \rightarrow \theta$, where Z is the input of the hypernetwork and φ are the weights. z is drawn from a probability distribution $\mathrm{p}_{\mathrm{noise}}$ and $\mathrm{p}_{\mathrm{data}}(\mathrm{x},\mathrm{y})$ is the data distribution of the given task.

The loss function of the main network can be written as $\mathcal{L}_{\mathrm{M}}(\theta|\mathrm{p}_{\mathrm{data}})$. Instead of training this function directly, we can expand it to the loss of the hypernetwork $\mathcal{L}_{\mathcal{G}}(\varphi|\mathrm{p}_{\mathrm{noise}},\mathrm{p}_{\mathrm{data}})$ of the combined network $\mathcal{M}(\mathrm{x};\mathcal{G}(\mathrm{z};\varphi))$ which we train for the weights φ.

To take the previously mentioned two criteria into account, the loss function can be specified to:

$$\mathcal{L}_{\mathcal{G}}(\varphi|\mathrm{p}_{\mathrm{noise}},\mathrm{p}_{\mathrm{data}}) = \lambda\mathcal{L}_{\mathrm{accuracy}}(\varphi|\mathrm{p}_{\mathrm{noise}},\mathrm{p}_{\mathrm{data}}) + \mathcal{L}_{\mathrm{diversity}}(\varphi|\mathrm{p}_{\mathrm{noise}}). \quad (2.12)$$

$\lambda > 0$ can be seen as a hyperparameter which balances the ratio between the two objectives accuracy and diversity. The accuracy term can be calculated as:

$$\mathcal{L}_{\text{accuracy}}(\varphi|\text{p}_{\text{noise}}, \text{p}_{\text{data}}) = \mathbb{E}_{\text{z}\sim\text{p}_{\text{noise}}}\mathcal{L}_{\text{M}}(\mathcal{G}(\text{z};\varphi)|\text{p}_{\text{data}}). \tag{2.13}$$

The diversity term is calculated through the entropy of the generated weights. But since an increase in entropy does not always increase the diversity of the output, Deutsch takes symmetry transformations into account. These trivial symmetry transformations include any composition of the following three : Scaling, logits' biases and permutations [5]. These symmetries are considered by a process called gauge fixing. The overall diversity is then calculated by estimating the overall entropy of the generated weights of a batch with the Kozachenko-Leonenko estimator [25]. The diversity loss $\mathcal{L}_{\text{diversity}}$ can be expressed as

$$\mathcal{L}_{\text{diversity}}(\varphi|\text{p}_{\text{noise}}) = -\mathbb{H}_{\text{z}\ \text{p}_{\text{noise}}}[\mathcal{G}(\text{z};\varphi)]. \tag{2.14}$$

With this approach Deutsch showed that the hypernetwork could not just learn the (locally) optimal weights of one main network, but a distribution of network weights. With the diversity term in the hypernetwork loss function, the created main network ensembles outperformed the individual networks and generalized superiorly.

This is in line with the results of Timur Garipov et al., who analyzed the loss surface of a classifier neural network and were able to show that local optima were connected via a low loss path with high accuracy. They demonstrated that the overall ensemble performance of randomly initialized networks was unmatched by other diversification methods [26]. Deutsch concluded that such ensembles could potentially be created by hypernetworks, since they are able to explore different modes of the loss surface.

2.3 Evolutionary Methods

The idea of evolutionary methods originates, just like the idea of neural networks, from nature. The key aspects are mutation and/or random combination of DNA/genes and selection, the survival of the fittest. The equivalent of DNA in the world of computer science, although not exclusively, is the parameter space of neural networks. The equivalent of selection are benchmark tests which select the best individuals/networks within each generation with respect to some predefined selection criteria [27].

In this scenario, the objective function of the RL problem (see equation 2.4) is considered to be a blackbox function, taking $\theta \in \mathbb{R}^{\text{d}}$ of a policy π_θ as input and outputting $\text{R}(\tau)$. The steps to train an evolutionary method like the genetic algorithm is, in its simplest form, as follows: First, a population of networks N is initialized randomly. For each individual of the population the performance in the given task is evaluated by some predefined measures. In a RL task this is mostly the maximum score reached, but also other measures like the diversity of a strategy can be incorporated in the evaluation. Due to this performance the best e individuals, called the elite, are selected. The weights of the best individual are kept unchanged. The weights of the other elites are permuted by randomly selecting one network of the elite and adding Gaussian noise to it. The process of modifying the weights of the elites is repeated until N-1 new networks have been created. This whole procedure is then repeated as a whole for a predefined number of iterations (called generations) or until one of the networks or all networks of the elite satisfy some final criteria. The pseudo code for a vanilla genetic algorithm

can be found in **Algorithm 3**.

Algorithm 3 Genetic Algorithm [27]

Input: mutation function ψ, population size N, number of selected individuals T (elite size), policy initialization routine ϕ, fitness function F
Draw $\mathcal{P}_i = \phi(\mathcal{N}(0, I)); i \in \{1, ..., N\}$ {initialize random DNN}
for g = 1,2, ... , G generations **do**
  Evaluate $F_i = F(\mathcal{P}_i^g)$
  Sort $\mathcal{P}_i^g$ with descending order by F_i
  select elite candidates $\mathcal{C}_i \leftarrow \mathcal{P}_{1...T}^g$
  for i in range N-1 **do**
    k = uniformRandom(1,T) {select individual from elite}
    $\mathcal{P}_i^{g+1} = \psi(\mathcal{C}_k^g)$ {mutate and create new population for next generation}
  end for
  add C_1 as true elite with out any permutations to $\mathcal{P}^{g+1}$
end for

One main advantage of evolutionary methods is their broad applicability. Since neural networks are universal function approximators almost any problem with an existing benchmark or score as a selection criteria can be dealt with. The downside of this method is that it can be quite inefficient.

Evolutionary methods have recently been applied to the field of reinforcement learning [27] and meta learning [28] and have proven to achieve state of the art results, especially when combined with gradient based methods. Many improvements have thereafter been developed [29, 30]. A field of particular interest in reinforcement learning have been methods, which do not solely focus on exploitation but rather encourage exploration. Edoardo Conti et al. computed, inspired by nature's drive towards diversity, the novelty of policies. This so-called Novelty Search (NS) encourages the exploration of not yet conducted behavior. Combined with an evolutionary strategy this approach proved to enhance the overall performance, since the algorithms were less likely to get stuck at local optima [31].

2.4 Diversity

Diversity is needed for a variety of tasks, yet can be interpreted, and therefore calculated in many different ways. In image classification, diverse network ensembles have shown to improve accuracy as well as out-of-distribution robustness [5, 26]. Important for the overall performance of ensembles is not only that the individual networks have high accuracy, but also that they make their mistakes in different parts of the data.[32] Stanislav Fort et al. show that such ensembles work best when their parameter space is diverse. They demonstrate that the decorrelation power of randomly initialized networks is unmatched by other diversity enhancing methods like Bayesian principles reviewed in their paper. These methods tend to solely explore areas of singular modes of the function space whereas random initializations are capable of exploring diverse modes [33]. To measure diversity resp. similarity, they compared the cosine similarity of weights of the neural network. The cosine similarity of two vectors A and B is calculated in equation 2.15.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} \tag{2.15}$$

This measure turns 1 if A and B are identical and 0 if the vectors are orthogonal. The parameters of neural networks can be turned into flattened weight vectors, so that the cosine similarity of the weight space of two networks v_1 and v_2 can be estimated with equation 2.15.

Deutsch also tries to create diverse networks form ensembles. Instead of measuring the diversity of the weights by calculating the cosine similarity, he tasks the entropy as a diversity measure. Since the overall entropy of weights can be increase by trivial symmetries like scaling or filterwise permutations, Deutsch performs a method called gauge fixing. The entropy therefore only increases when the weights are *essentially different*, and therefore no trivial symmetry transformation exists.[5] The entropy H of the weights is estimated by the Kozachenko-Leonenko estimator [25] (see 2.16).

$$H = \psi(N) + \frac{d}{N}\sum_{i=1}^{N}\log(\epsilon_i) \tag{2.16}$$

Here, ψ is the digamma function, the logarithmic derivative of the gamma function $\Gamma(N)$, d is the dimension of the samples, in this case the dimension of the input of the hypernetwork (chapter 2.2), and ϵ_i is the distance from one flattened weight vector θ_i to its nearest neighbor in the set of sampled weight vectors [5, 34].

In the field of RL exploration can be improved through effective diversity [3]. Jack Parker-Holder et al. try to enhance the overall performance of RL-agents by boosting exploration. Having a population of agents and a multi-objective loss function increases diverse behavior. However, the differences between the policies of such populations remain a difficult property to measure. Typically the contribution of each individual to the overall population reward-diversity objective is calculated. This may lead to constant switching of behavior between the individuals and prevent single agents from exploiting promising behavior further [35]. To tackle this problem, Parker-Holder et al. introduced a method called *Diversity via Determinants* (DvD).

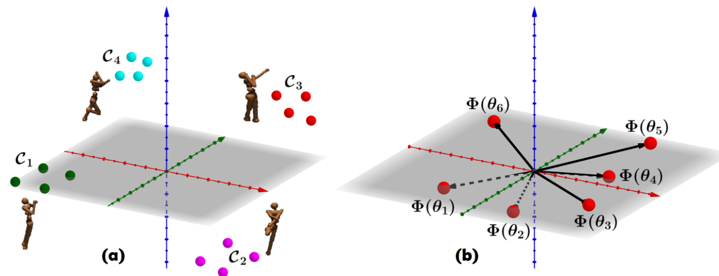


Figure 2.7: (a) Pairwise distance: populations of agents split into cluster with agents within one cluster exploring similar policies. (b) Determinants: Embedded policies $\phi(\theta_i)$ lie in a hyperplane.[3]

Instead of restricting the policy update $\pi_{\theta_{t+1}}$ by some NS objective, they consider action-

based behavior embeddings. The authors mentioned before investigated diversity in the weight space of neural networks. This isn't reasonable for RL-agents which are supposed to explore diverse strategies. Instead, the diversity of the action space should be evaluated. To do so *Behavioral Embeddings* (BE) are introduced which are defined as follows:

Behavioral Embeddings: Let θ^i be a vector of neural network parameters encoding a policy π_{θ^i} and let $\mathcal{S}$ be a finite set of states. The BE of θ^i is defined as: $\phi(\theta^i) = \{\pi_\theta(\cdot|s)\}_{s\in S}$.[3]

This means that for $\phi : \theta \rightarrow \mathbb{R}^l$ two policies are identical when for each state the same actions are chosen (equation 2.17). $l = |a| \times N$, where *a* is the dimensionality of the action and N the number of states.

$$\phi(\theta^i) = \phi(\theta^j) \Leftrightarrow \pi_{\theta^i} = \pi_{\theta^j} \tag{2.17}$$

θ^i and θ^j represent the flattened weight parameters of the networks associated with the policies. The embeddings of such policies are approximated by

$$\phi^*(\theta_i) = \mathbb{E}_{s\sim\mathcal{S}}[\{\pi_{\theta^i}(\cdot|s)\}] \tag{2.18}$$

This means that all the states are drawn form a joint replay buffer. So all agents are trained with the same states, even though they explore the environment independently.Two such policy-embeddings can be compared using a kernel function defined as $k(x_1, x_2) \leq 1$. A popular choice is the squared exponential kernel, but any kernel can be chosen. Transferred to the behavioral embeddings, the similarity between two policies can be defined as $k(\phi(\theta^i), \phi(\theta^j)) = 1 \Leftrightarrow \pi_{\theta^i} = \pi_{\theta^j}$. Polices can be considered to be orthogonal when $k(\phi(\theta^i), \phi(\theta^j)) = 0$.

To estimate the overall diversity of a population of agents, Jack Parker-Holder et al. defines the populations diversity as follows:

Population Diversity: Consider a finite set of M policies, parameterized by $\theta = \{\theta_1, ..., \theta_M\}$, with $\theta_i \in \mathbb{R}^d$. We denote $\mathrm{Div}(\Theta) = \det(k(\phi(\theta^i_t), \phi(\theta^j_t))^M_{i,j=1}) = \det(\mathbf{K})$, where $\mathbf{K} : \mathbb{R}^l x \mathbb{R}^l \rightarrow \mathbb{R}$ is a given kernel function. Matrix **K** is positive semidefinite since all principal minors of det(**K**) are nonnegative.[3]

From a geometric perspective this means that the determinant of the kernel matrix represents a parallelepiped plane spanned by the feature maps of the corresponding kernel. By maximizing the determinant the volume of this feature space is effectively "filled". Since in the case of using BE the feature space can be considered to be equivalent to a behavior space, this also maximizes the diversity of the actions [36]. For a detailed derivation of the concepts above, see the Appendix of [3].

Having defined such a new diversity measure, the objective function can be defined as

$$J(\Theta_t) = \sum_{i=1}^{M} \mathbb{E}_{\tau\sim\pi_{\theta^i}}\Big[R(\tau)\Big] + \lambda_t \mathrm{Div}(\Theta_t). \tag{2.19}$$

Parker Holder et al. show that this approach finds the different modes of solutions in multi-modal environments. This is very similar to equation 2.12. The first part of the sum represents the individual rewards, whereas the second part takes the diversity of the population into

account. Though this objective was formulated in the context of ES, it can be transferred to the loss function of hypernetworks.

A difficult topic remains the choice of λ. It highly effects the overall performance of the algorithm. If it is chosen too low, no diversity is found, since the selection pressure is too low. Chosen too high, the population won't reach local optima but will only behave in a more and more diverse way.

To tackle this problem, multi-armed bandits [37] are introduced as proposed by Jack Parker-Holder et al.. This enables the algorithm to favor the different objectives, reward and diversity, at different stages of the optimization process by shifting the value of λ.

In the next chapter the methods and concepts used in this thesis will be introduced. The focus will lie on the implementation of the hypernetwork as well as the measurement of diversity. The concepts developed will expand the approaches covered in this chapter.

3 Methods and Concept

3.1 Hypernetworks for Reinforcement Learning

Instead of training the neural network of a RL algorithm directly, a hypernetwork can be used to generate the weights of such networks. Though normally a hypernetwork is only used to generate the weights of one main network (or a batch of the main network's weights), it can also be designed to generate the weights of many different networks.

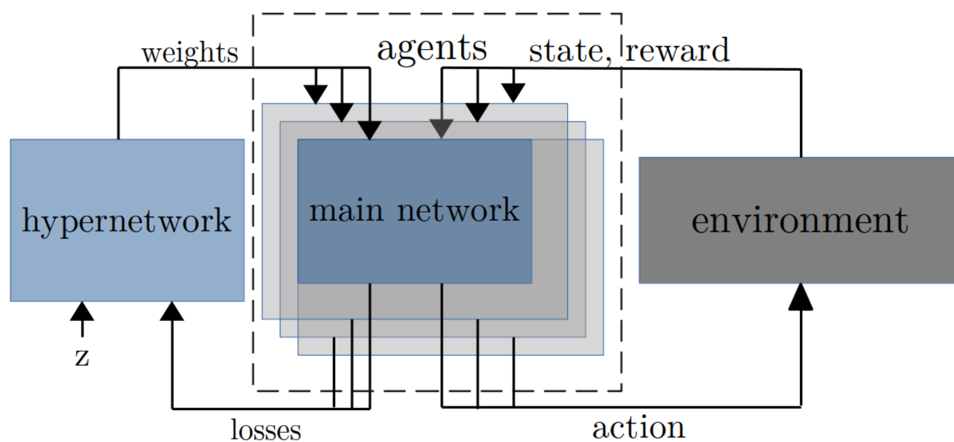


Figure 3.1: With a random vector z the hypernetwork (left) creates a set of weights of agents (middle) which interact with the same environment (right), but are independent from one another.

In figure 3.1 the overall structure of the approach can be seen. A hypernetwork receives a

batch of random vectors z as input and generates the weights of the networks of the agents. The agents then interact with the environment by taking actions and receiving states and rewards. With these trajectories the losses of the agents are calculated and the hypernetwork is updated. Then a new batch of random vectors z is drawn and the next iteration starts.

In the case of a DDQN, the weights of the target network are updated every n training steps and are set equal to the weights of the trained online network. This means that only one set of weights has to be generated by the hypernetwork: the weights of the online network. These weights can then, just like it is the case in 'vanilla' DDQN training, be taken to update the weights of the target network.

For the A2C, the hypernetwork has to create two different sets of weights for actor and critic. Depending on the implementation of the A2C, actor and critic network can share the first layer(s) and only differ in the hidden and/or output layer(s). This can be helpful to decrease the number of parameters needed to be trained. The hypernetwork will create some shared weights for both networks as well as distinct ones for either actor or critic (see chapter 3.1.2).

The hypernetwork used will take a random vector z of dimension d sampled from a uniform distribution between -1 and 1 as input. For each vector $\mathrm{z_i}$ the hypernetwork generates one set of weights φ_i for the main network with $\mathrm{i} \in \{1, .., \mathrm{n}\}$, where n is the so-called hypernetwork batch size. With each of these generated main networks an episode in the game environment is played. In the case of DDQN, these episodes are saved in separate replay buffers for each agent. For the A2C no replay buffer is needed.

The hypernetwork creates a batch of n different agents. Each one of these agents interacts with the same environment, but every agent is trained only with it's own transitions. This enables the creation of a population of diverse agents, similar to the population used in ES-algorithms [31].

3.1.1 Loss Function

Following the approach of Deutsch and Parker-Holder et al., the loss function of the hypernetwork $\mathcal{L}_\mathcal{G}$ is split into two terms

$$\mathcal{L}_\mathcal{G} = \mathcal{L}_{\text{agents}} + \lambda \mathcal{L}_{\text{divserity}} \tag{3.1}$$

The $\mathcal{L}_{\text{agents}}$ corresponds to the sum of the loss functions of either the DDQN's or the A2C's (see 2.3 and 2.9).

In contrast to the approach of [5] and [26], in the case of RL, diversity of weights isn't the objective. Even though building diverse ensembles of networks might also be applicable to RL problems, this won't be considered here. Instead, a different diversity measure is used, namely the diversity in action space. The BE as proposed by [3] will be used as a basis for two different evaluation methods of diversity.

3.1.1.1 Cosine Similarity

One way to calculate diversity is the cosine similarity (see equation 2.15). When doing so for a batch of agents generated by a hypernetwork, the states used as inputs have to be the same for all main networks in order to be able to estimate the diversity of the actions. During training this is normally not the case, since all agents are trained with their own trajectories. At

each gradient update of the hypernetwork, states are sampled from the state space $\mathcal{S}_i$ of one main network M_{θ_i} of the hypernetwork batch. These states are then used as input for all main networks and the actions are used to calculate a cosine similarity matrix of all main network pairs. Note that only a pairwise similarity estimation is possible with equation 2.15. The values of the matrix are then summed up and yield the value of the diversity loss $L_{diversity}$ of the batch.

3.1.1.2 Diversity via Determinants

Another way to estimate the diversity between the actions of agents is DvD [3] as has been shown above. Instead of using the cosine similarity to estimate the diversity term in equation 3.1, the diversity measure from equation 2.19 can be applied. The diversity can be defined as

$$\mathrm{Div}(\Theta) = \det(\mathrm{k}(\phi(\theta_t^i), \phi(\theta_t^j))_{i,j=1}^{M}) = \det(\mathbf{K}) \tag{3.2}$$

and can be set to $\mathcal{L}_{\text{diveristy}}$, where $\mathbf{K}$ is the kernel matrix. The embeddings ϕ^i are the Q- or probability values for the actions of the agents M. As a kernel k, the squared exponential kernel is chosen (see equation 3.3). x_1 and x_2 are the embedding vectors and l is some length scaling set to 1.

$$\mathrm{k}_{\mathrm{SE}}(\mathrm{x}_1, \mathrm{x}_2) = \exp\Big(- \frac{\| \mathrm{x}_1 - \mathrm{x}_2 \|^2}{2\mathrm{l}^2} \Big) \tag{3.3}$$

In order to use equation 3.2 as $\mathcal{L}_{\text{diversity}}$ in the loss function of the hypernetwork, it must be possible to calculate analytic gradients of $\det(\mathbf{K})$. This is possible as has been proven by **(author?)**. The hypernetwork can then be updated using automated differentiation. The proof of the lemma below can be found in the Appendix of [3].

Lemma 3.1. *The gradient of* $\log(\det(\mathbf{K}))$ *with respect to* $\Theta = \theta_1, ..., \theta_M$ *equals:* $\nabla_\theta \log(\det(\mathbf{K})) = -(\nabla_\theta \psi(\theta))(\nabla_\psi \mathbf{K})\mathbf{K}^{-1}$, *where* $\varphi(\theta) = \varphi(\theta^1)...\varphi(\theta^M)$.

The pseudo code of the hypernetwork can be seen in **Algorithm 4**. For detailed information about the implementation see Appendix 5 and 5.

3.1.2 Architecture

The overall architecture of the hypernetwork is the same as in [5]. The main network M has *m* layers and *h* filter/neurons or subgroups of neurons. $\theta_{l,i}$ are the weights corresponding to the *i*'th neuron/filter/subgroup of the *l*'th layer.

Because the parameter size of the hypernetwork H would scale badly with the size of the main network, if it would simply be a multi layer perceptron (MLP), where each output neuron corresponds to a weight parameter of the main network, a generator structure is chosen instead (see figure 3.1.2).

A *d*-dimensional input vector *z* is taken as input of the extractor network *E*. This extractor outputs for each layer embedding $e_{m,h}$. One embedding vector corresponds either to a single neuron or a subgroup of neurons in the layer. Depending on the size and the needed resolution the size of the encoding as well as the size of the subgroup is chosen. This vector is taken iterativly as input of the weight generator networks W_m. Each layer of the main network has its corresponding weight generating network, which outputs for each part of the encoding

Algorithm 4 Hypernetwork

Initialize hypernetwork $\mathcal{G}(z; \varphi)$ and environments
for each iteration **do**
  draw random vector z_n with n as hypernetwork batch size
  create n sets of weights of agents with weights $\{\theta_1, ..., \theta_n\} = \mathcal{G}(z_n; \varphi)$
  for each agent of batch **do**
    play episode of the game and store transitions in $\mathcal{D}_i$
  end for
  for n in range minibatches **do**
    sample states s_i of size minibatch from $\mathcal{D}_i$
    calculate the losses for $\mathcal{M}_{\theta_i}(s_i)$
    estimate the diversity of the batch $\mathcal{L}_{diversity}$
    perform gradient update $\mathcal{G}(z_n; \varphi)$
  end for
end for

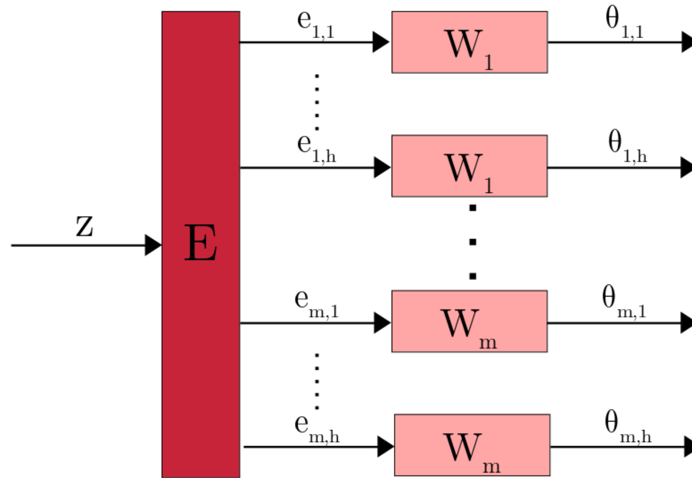


Figure 3.2: The Extractor E takes the random vector z as input and outputs the embedding vectors e. These vectors are fed to the weight generators $W_1 ... W_m$ which output the weights for the main network θ. The Extractor outputs several embeddings for each weight generator. Each weight generator corresponds to one layer, so for each layer of the main network, the same weight generator is used.

vector $e_{m,h}$ the weights. With this approach it is possible to reduce the size of the hypernetwork significantly.

For the DDQN the architecture is quite simple. For each layer of the DDQN online network one weight generator is created. For the A2C the same weight generator is used for the first layer of the actor and the critic, but the embeddings by the extractor for the layers differs. For the other layers distinct weight generators are used.

Weight initialization has been widely studied for neural network, but almost no research has been done on the weight initilization of hypernetwork [38]. Weight initilization methods like

the the glorot-method [39] are widely used, but fail to generate weights for hypernetwork of an adequate scale. This is because normally the weights are sampled uniformly from a distribution which highly depends on the size of inputs of the layer. For hypernetworks this leads to bad results. Instead, the weights of a hypernetwork are drawn according to a normal distribution from the glorot-method.

3.2 Evolutionary Strategy

An evolutionary method like the genetic algorithm can be used to find diverse playing agents. In evolution based algorithms the selection criteria is crucial for the overall outcome of the population. As has been described in chapter 2.3, an evaluation of the population is performed after each generation and an elite is selected. The selection criteria is often the achieved score of the individuals. This follows the idea of 'survival of the fittest'.

In evolutionary methods, as opposed to gradient methods, the network is considered to be a black box. Salimans et al. [40] train the networks by applying additive Gaussian noise to the parameters of the networks. The gradient is estimated by taking the sum of the sampled perturbations and weigh it by the reward. The weighting can be expanded by a diversity term to account for novelty seeking behavior [31].

$$\theta_{t+1}^{m} = \theta_{t}^{m} + \eta \frac{1}{k\sigma} \sum_{i=1}^{k} [R_i^m + \lambda \mathrm{Div}_t(i)] g_i^m \tag{3.4}$$

This leads to equation 3.4, which shows the parameter update of a population of size M. A set of agents $\Theta_t = \{\theta_t^i\}_{i=1}^{M}$ at iteration t is considered. Gaussian perturbations of size $\{g_i^m\}_{i=1,...,k}^{m=1,...,M}$ are drawn and the set $\{g_1^m, ..., g_k^m\}$ is sent to the m'th worker. This worker then evaluates the reward of the agent θ_t^m for each of the k permutations (R_i^m). A second partitioning is used to calculate $\mathrm{Div}_t(i)$: Subsets of $\mathcal{D}_i = \{g_i^1, ..., g_i^M\}$ are created and the diversity is calculated by

$$\mathrm{Div}_t(i) = \mathrm{Div}_t(\phi(\theta_t^1 + g_i^1), ..., \phi(\theta_i^M + g_i^M)). \tag{3.5}$$

The contribution of the entire subset of $\mathcal{D}$ is therefore considered, instead of only one individual vector g_i^m. In equation 3.4 η is the learning rate and σ is a smoothing parameter. This leads to **Algorithm 5**.

Algorithm 5 Evolutionary Strategy (DvD) [3]

Input: population size M, fitness function F, diversity function D
Initialize i random DNNs: $i \in \{1, ..., N\}$
for g = 1,2, ... , G generations **do**
  Draw random vectors $\{g_i^m\}_{i=1,...,k}^{m=1,...,M}$
  Evaluate score of the agents $F_i = F(\theta_i^g + g_i^m)$
  Calculate diversity for g_m: $\mathrm{Div}_t(i) = \mathrm{Div}_t(\phi(\theta_t^1 + g_i^1), ..., \phi(\theta_i^M + g_i^M))$
  Estimate gradients and update weights $\theta_{t+1}^m = \theta_t^m + \eta \frac{1}{k\sigma} \sum_{i=1}^{k} [R_i^m + \lambda \mathrm{Div}_t(i)] g_i^m$
end for

A simpler approach follows the idea of [27] doesn't keep a static population of agent and tries to estimate the gradients with perturbations, but rather generates a mutated new population after each generation. Essentially, the approach described in **Algorithm 3** is further developed:

A population of agents is evaluated by some fitness function and a elite is selected. But instead of randomly mutating the elite now until the initial population size is restored, several random vectors are evaluated in terms of diversity. For each random vector by which the elite members are mutated the overall impact on the diversity of the whole elite group is measured via the DvD approach. Then the elites are mutated only with the random vectors which scored highest in terms of diversity. Score and diversity are considered when selecting and mutating the population. This can be seen in **Algorithm 6**:

Algorithm 6 Genetic Algorithm

Input: population size N, number of selected individuals T (elite size), policy initialization routine ϕ, fitness function F
Draw $\mathcal{P}_i = \phi(\mathcal{N}(0, I)); i \in \{1, ..., N\}$ {initialize random DNN}
for g = 1,2, ... , G generations **do**
  Evaluate $F_i = F(\mathcal{P}_i)$
  Sort $\mathcal{P}_i$ with descending order by F_i
  select elite candidates $\mathcal{C}_i \leftarrow \mathcal{P}_{1...T}$
  for i in range mutation number m **do**
    Draw g_i and estimate the impact on the elite members
    $\mathrm{Div}(i) = \mathrm{Div}(\phi(\mathcal{C}^1 + g_i), ..., \phi(\mathcal{C}^T + g_i))$
  end for
  Sort g_m with descending order by Div_i score and select best mutation vectors g^*
  for i in range N **do**
    Add g^* randomly to elite $\mathcal{C}_i$
  end for
end for

In the next chapter the experiments used to evaluate the proposed new concepts will be described and analyzed. The new algorithms will be tested in a variety of tasks to prove the applicability of the hypernetwork approach to reinforcement learning problems.

4 Experiments

In this thesis the hypernetwork approach is evaluated in three experiments. First, the image classification experiment from Deutsch's paper will be reproduced. Second, a simple graph game experiment will serve to evaluate the applicability of the new diversity measures to the overall hypernetwork approach. Additionally, the hypernetwork will be bench marked against two population based methods as proposed in chapter 3.2. Lastly, the hypernetwork will be tested in a challenging imperfect information game. The code can be found here:

https://github.com/GeorgMiller/Hypernetworks

4.1 Image Classification

As a first experiment a simple image classification task is performed. As a dataset the MNIST dataset is chosen. The classifier network (main network) will be the same as in Deutsch's paper [5]. All implementation details can be found in the appendix.

The input of the hypernetwork z is for the classification task uniformly drawn from $[-1, 1]$. The size of z is 300. The size of the vector seems big compared to other generative methods like GAN's, but since this size was chosen by Deutsch, it will be kept for the image classification experiment. The loss gradients are estimated using batches of size 32. For each vector z of the batch, 32 images (minibatch size) are classified and the loss is calculated. The hypernetwork is trained for a total of 10.000 batches. The hyperparameter λ is chosen to be 10^3, 10^4 and 10^5. The hypernetwork has a total of 516.040 trainable weights, the main network a total of 20.018 weights.

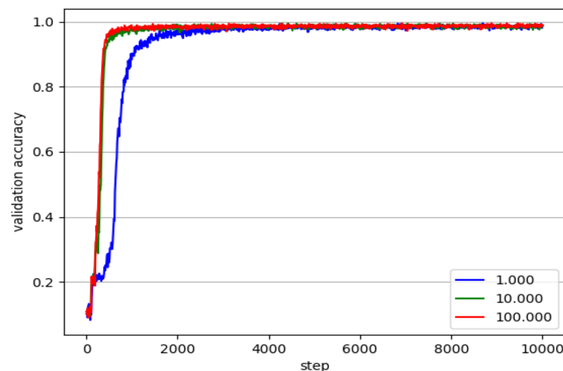


Figure 4.1: Hypernetwork validation set accuracy for different λ

4.1.1 Accuracy

The training of the hypernetworks on the MNIST dataset can be seen in figure 4.1. The validation set accuracy of randomly sampled classifier networks from the hypernetwork batches quickly reaches around 98%. The parameter λ for the later experiments is chosen to be 10.000

since the accuracy as well as the training time doesn't change by a large margin between 10.000 and 100.000.

4.1.2 Diversity

One way to show that the hypernetwork doesn't just create noise around a singular optimal solution but instead finds diverse local optima of the loss landscape is to construct a path between two vectors z_1 and z_2. Two different paths can be generated: directly by $\{\mathcal{G}(z_1;\varphi)t + \mathcal{G}(z_2;\varphi)(1-t)|t \in [0,1]\}$ which combines the generated weights of the two vectors linearly after generation and an interpolated path $\{\mathcal{G}(z_1 t + z_2(t-1);\varphi)|t \in [0,1]\}$ of the hypernetwork input. The direct path would only then have high accuracy, if the diversity is achieved solely by adding isotropic noise around a singular optimal vector. As can be seen in figure 4.2, after 100 batches the weight path constructed by the hypernetwork already starts to outperform the linear weight combination, but the solutions on the loss surface still lie close to each other. After training for 10.000 batches, the hypernetwork constructs a high accuracy path between two solutions, which are separated by a low accuracy valley in weight space. Therefore, the diversity of the generated weights isn't due to trivial operations. This coincides with the results of Deutsch. It shows that the hypernetwork generates diverse samples from the distribution of optimal weights of the main network.

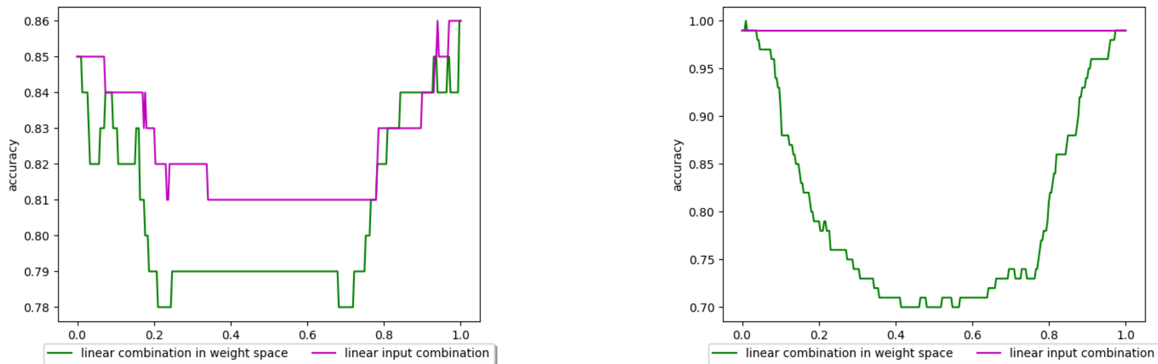


Figure 4.2: Linear and interpolated path in weight space between two random vectors in the beginning (left) and after training (right).

The diversity of the generated weights can be compared to randomly initialized and then independently trained networks. In figure 4.3 the cosine similarity of the flattened weight vectors of 32 randomly initialized classifier networks are compared with a batch of networks generated by the hypernetwork. During training, the hypernetwork first generates similar weights. After 10.000 epochs, the solution space of the generated networks becomes more diverse, but does not match the diversity score of independently trained networks. In future experiments, the training time could be increased to evaluate if the hypernetwork could find as diverse solutions as independently trained networks. Compared to other diversification strategies as analyzed by Garipov et al., the hypernetwork approach creates a more diverse set of weights [26].

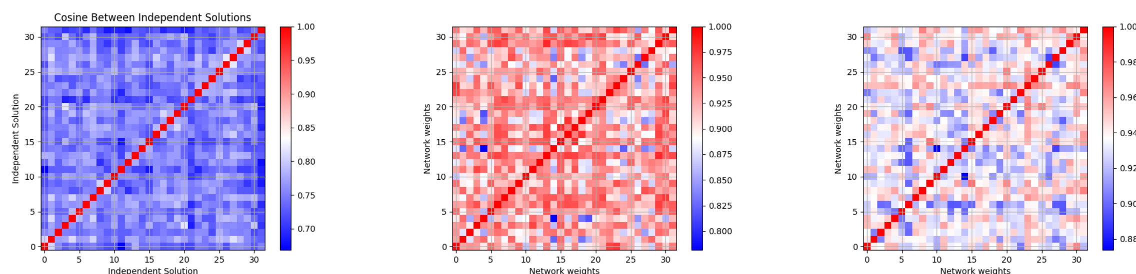


Figure 4.3: Cosine similarity of randomly initialized trained classifier network weights (left). Cosine similarity of generated weights by the hypernetwork during (middle) and after training (right).

The experiments show that a single hypernetwork is capable of generating a variety of diverse samples from the distribution of weights of main networks. The diversity measure can easily be modified by manipulating or exchanging the diversity term in the loss function. This is an easy and uncomplicated strategy and it seems plausible to apply this method to other research fields.

4.2 Graph Game

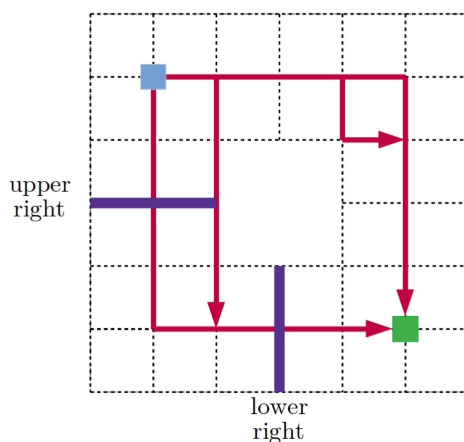


Figure 4.4: Graphical illustration of the graph game with starting point (blue) and goal (green). The two purple thresholds *upper right* and *lower right* indicate which path the agent takes. The four red paths are diverse optimal policies by an agent.

To evaluate the hypernetwork approach in a RL environment, the A2C and the DDQN are tested in a simple graph game. The graph game environment considered in this experiment is a simple squared grid of size 7x7. In the beginning of an episode the agent starts in the upper left corner at coordinates (1,1) and can move along the grid one node at each time step. It can select between four actions: Up, down, left right. In the middle of the grid, the connections between the nodes are removed so that an obstacle emerges. Whenever the agent tries to take an action which isn't permitted due to missing links, it will remain in the same state as before.

In the lower right corner the goal-node is located at (5,5). The agent will receive a reward of 1 if it reaches this node and the episode ends. The episode also ends if a maximum number of 20 moves is reached. For each move the agent receives a small penalty to enforce finding an optimal strategy. A graphic of the game can be seen in figure 4.4 as well as three optimal strategies.

In figure 4.4 two thresholds can be seen. They are used to determine if an agent decides to either take the right or the left path. Since the maximum number of moves is 20, the agent can only cross both thresholds and reach the goal if it takes the right path. If it only crosses one, the agent might have switched its decision.

The training of an A2C and a DDQN can be seen in figure 4.5. The easy RL-task is mastered in less then 1000 episodes by both of them, but the A2C doesn't reach an optimal equilibrium. Instead, the agent fluctuates around the optimal solution without stabilizing. The DDQN on the other hand needs more training time but reaches an optimal solution. Therefore, the DDQN agent is chosen for the future experiments with the hypernetwork as well as the evolutionary methods.

To evaluate the hypernetwork, a random vector z of size 4 is drawn uniformly between -1 and 1. The losses are calculated as described in chapter 3. To estimate the diversity, the cosine loss as well as the DvD loss is used. The exact details of the architecture of the hypernetwork can be found in appendix 5.

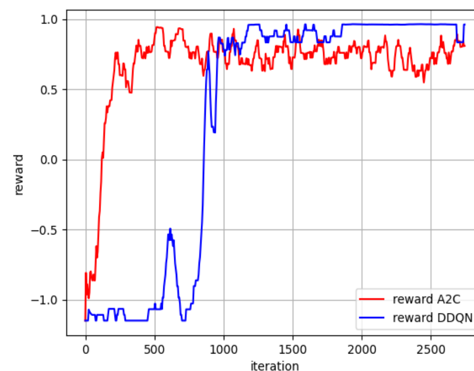


Figure 4.5: The normal training of A2C and DDQN.

The hypernetwork batch size is chosen to be 2 for most of the experiments, but also an experiment with larger batch sizes is conducted. Each of the agents of a batch plays one round of the graph game and the transitions are stored in separate replay buffers. Then the losses of all the agents on their transitions is calculated and summed up. Thereafter 15 states are sampled from the replay buffer of one randomly picked agent and fed as input to all the agents. The resulting actions are taken as the embedding vectors to estimate the diversity.

As can be seen in figure 4.6 a normally trained agent decides to choose only one direction (left), without reaching the goal. After 1000 episodes the agent always goes left and manages after an additional 1000 episodes to always reach the target (note that the optimal score is 0.96). The hypernetwork, when trained without the diversity loss simply with the same objective function as the normal DDQN and with a batch size of one, shows a similar behavior. After around 400 episodes the agent starts to mainly go right without reaching the goal. After 600 episodes the agent reaches almost every time it goes right the target. The solution of the hypernetwork is not as stable as the one found by the normal DDQN due to the random vector as input. Longer training times only partially decrease this instability. Similar to the A2C, the hypernetwork doesn't seem to find a stable optimal solution without fluctuations.

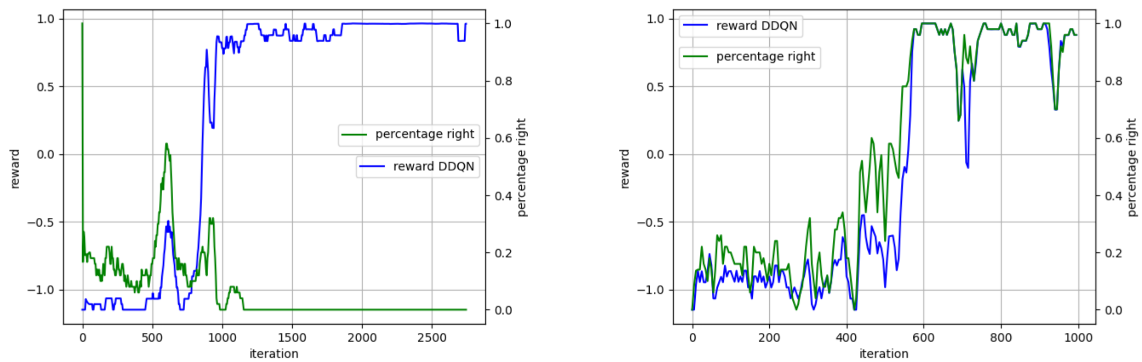


Figure 4.6: Reward and percentage of going right of the DDQN network (left) and the DDQN-hypernetwork without a diversity loss therm (right).

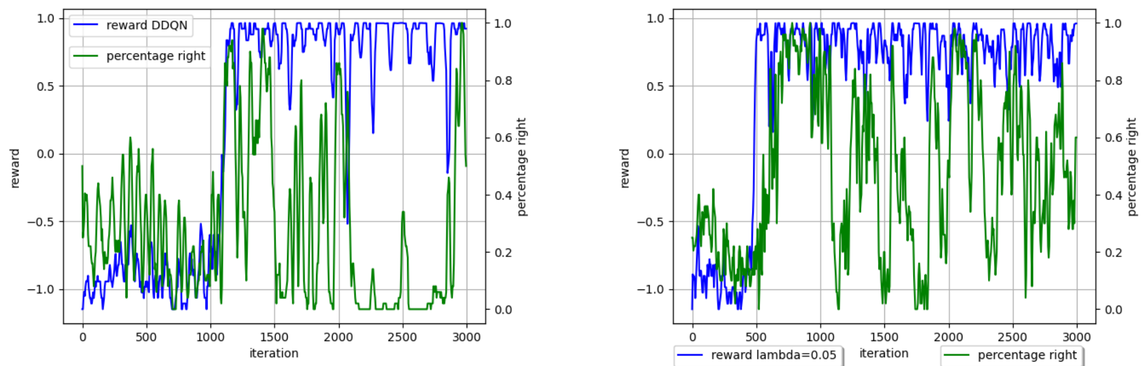


Figure 4.7: Reward and percentage of going right of the hypernetwork trained with the cosine loss (left) and $\lambda = 0.8$ and the DvD loss (right) with $\lambda = 0.05$.

Training with Cosine Similarity: The training of the hypernetworks with the cosine loss with a value for λ of 0.8 can be seen in figure 4.7. Again, in the beginning the percentages for going right show moderate values while the generated main networks are still exploring. After around 1100 iterations the generated agents manage to reach the target most of the time, but at the same time the networks do not always go right or left, but instead the percentage fluctuates strongly. This indicates that different solutions are generated for the randomly drawn input vectors.

Training with Diversity via Determinants: In figure 4.7 the training of the hypernetwork with the DvD loss can be seen. The parameter λ was chosen to be 0.05, but hyperparameter tuning as well as an adaptive mechanism as proposed in [3] may lead to better results. The hypernetwork finds an, albeit unstable, optimal solution after 500 iterations. The percentage of going right highly fluctuates without apparent influence to the overall score. The estimated

mean is around 60%. Compared to the cosine loss, the DvD loss shows faster learning and a more stable score. Also it seems that the DvD loss causes the hypernetwork to explore the two different solutions in a more balanced way.

In figure 4.8 the percentage of going right is plotted together with the DvD loss. Instead of 2 the batch size is increased to 8. The higher batch size leads to a slower decrease of the loss and a higher fluctuation in the choice of direction. This is in line with the results of Parker-Holder et al., which showed that the batch size should be picked in the same scale as the solution space.[3]

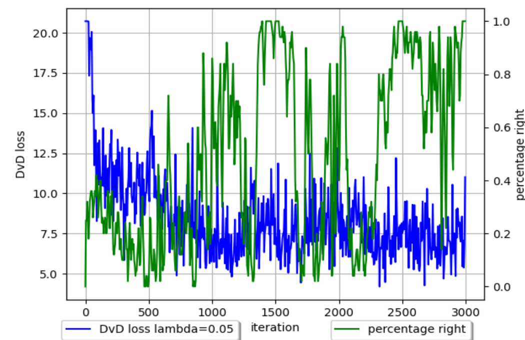


Figure 4.8: Hypernetwork DvD loss and percentage of going right with batch size of 8 and $\lambda = 0.05$

Genetic Algorithm and Evolutionary Strategy: The hypernetwork can be compared against other diversity enhancing algorithms like population based learning. In figure 4.9 two approaches introduced in chapter 3.2 are tested as a comparable baseline for the hypernetwork approach. The Genetic Algorithm, as described in **Algorithm 6** is trained with a population size of 20 and an elite size of 4. Each generation first evaluates all 20 agents for 2 episodes. The 50 mutation vectors are randomly added to the elite and the diversity enhancement is measured. The 20 vectors leading to the highest population diversity are then used to recreate the population of size 20. The population is trained for 3000 generations, but this is not comparable to the computational time of 3000 episodes of the hypernetwork. Each generation needs more then 20x more episodes played than the hypernetwork. As can be seen in the graph, the GA fails to find a diverse set of agents. Even though the agents reach an optimal score after 2250 generations, the population itself doesn't yet behave diverse after 400 generations.

The second population based algorithm, the ES, is able to find an optimal and diverse solution. The population size of the ES approach described in **Algorithm 5** is 6. The gradient is approximated with a combination of the DvD loss and the reward (see equation 3.4). The learning rate is chosen to be 0.01 and in each generation 10 random vectors are evaluated for each agent in the population. The population is trained for 3000 generations, which need more then 6x the computation time of the hypernetwork approach. After 200 generations the first member of the population finds the first optimal solution by going right. The second agent of the population find the second solution after 2200 generation by going left. The population is hence capable of finding both diverse solution of the graph game.

The ES and the hypernetwork both find the diverse solutions while normally trained agents and the GA approach fail to do so. The hypernetwork surpasses the evolutionary approach in terms of computation time. While the ES needs more then 2000 generations to find both solution the hypernetwork only needs around 1100 iterations. The speedup is much larger than the factor of 2, since one generation equals approximately 6 iterations.

The experiments of the graph game have shown that the hypernetwork approach is applicable to RL-problems. Not only does the hypernetwork find optimal solutions of the given task, they are also diverse when a diversity loss term is added to the loss function.

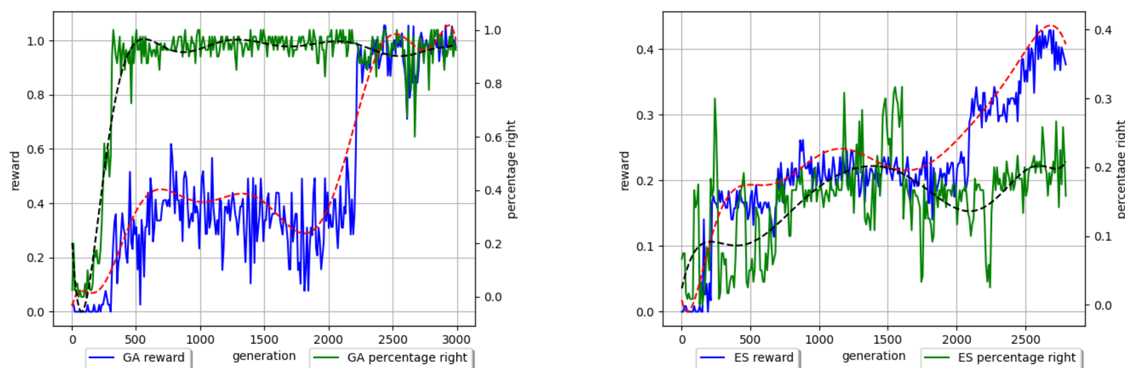


Figure 4.9: Reward and percentage of going right of the genetic algorithm (left) and the evolutionary strategy algorithm (right).

4.3 Imperfect Information Game

Before choosing an imperfect information game, different requirements need to be considered:

1. One of the main problems of research in AI is the hardware needed to get state of the art results. Most of the papers reviewed used several servers for days or weeks for training. Therefore competing against these algorithms doesn't seem reasonable.

2. The chosen game should be complex enough to have different tactics in order to see if the diversity in weight space correlates with different playing styles.

Considering the above criteria, DouDizhu, a chinese card game, is chosen. The rules of the game are noted in the box below.

The framework of the game was taken from the RLCard project [41], but some modifications were made. One main challenge for RL in DouDizhu is the large action space. The number of possible combinations is 27.472, an impractical output of a neural network. Therefore, some simplification has to be made. The RLCard framework performs action grouping and reduces the amount of possible actions to 309. This is still quite a large action space, but reasonable. The action grouping can be seen in table 4.1. If the action selected by an agent isn't possible because the agent doesn't contain the necessary cards, the framework randomly selects a playable action. Note that selecting the action 'pass' is mostly, but not always possible. Therefore 'pass' cannot be selected as a default action. Once an agent gets rid of all of its cards, the episode ends and a reward of 1 for the winning party and a reward of 0 for the loosing one is set.

A simplification of the game in the RLCard framework is, that the bidding phase doesn't exist. Instead, the cards of each player are evaluated heuristically before each game and the player with the best hand is selected as landlord automatically. Therefore, it is possible to determine whether the agents should (always) play as landlord or peasants.

DouDizhu is a chinese card game which is played with a 54-card deck including a red and a black joker. The rank of the cards from high to low is: **red joker, black joker, 2, A, K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3**. The color of the cards is unimportant. In the beginning of each game every player receives 17 cards. The remaining three cards are put face down in the middle.
To determine the roles (Landlord, Peasant, Peasant) the player who received the first card can make a bid of 1, 2 and 3 or pass. The next players in the round can then either bid higher or pass. If all players pass, the cards are shuffled again. The player with the highest bid becomes landlord and takes the three cards in the middle to his hand.
The landlord starts the game by playing a combination of cards (see table 4.1). The player to his left can either play a higher card, a higher combination of cards with the same number of cards and the same combination or the player can pass. The round continues in the same manner until two players successively play pass. The played cards are then put aside and the player who played the last combination can start the new round with any combination. There are two exceptions: A rocket (two jokers) can beat any combination and can always be played (if it's the player's turn). Also a bomb (four of a kind) can beat any combination, except for a rocket or a higher bomb.
The player who first gets rid of all of his cards wins the game. Depending on the bid at the beginning of the game, 1, 2 or 3 points are received if the landlord won. If one of the peasants won, both receive half of the points as reward. If a bomb or a rocket is played during the game by either player, the bidding points are doubled each time and distributed accordingly to the winning player(s).

Type	Number of Actions	Number of actions after Abstraction	Action ID
Solo	15	15	0-14
Pair	13	13	15-27
Trio	13	13	28-40
Trio with single	182	13	41-53
Trio with pair	156	13	54-66
Chain of solo	36	36	67-102
Chain of pair	52	52	103-154
Chain of trio	45	45	155-199
Plane with solo	21822	38	200-237
Plane with pair	2939	30	238-267
Quad with solo	1326	13	268-280
Quad with pair	858	13	281-293
Bomb	13	13	294-306
Rocket	1	1	307
Pass	1	1	308
Total	27472	309	

Table 4.1: Action grouping for DouDizhu

As can be seen above, the actions of the game are encoded into an array of length 309. The state encoding on the other hand is more complex. Some changes had to be made to the orig-

inal state encoding of the RLCard framework, because it only contained a minimal amount of information necessary to play the game. This led to poor performance of the playing agents. In cooperation with Jing Li a new state encoding was developed. It consists of 9 feature planes. Each of these one-hot encoded planes is of shape 5x15 (see table 4.2). The five rows represent the amount of cards (starting in the first row with zero, up to four in the last row) held by the player, the fifteen columns represent the card values from '3' to 'RJ' (going left to right). In the example in table 4.2 one 3, two 4s and four 8s are encoded. Each of the seven feature planes encodes distinct information, as can be seen in table 4.3.

0	0	1	1	1	0	1	1	1	1	1	1	1	1	1
1	1	0	0	0	1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

Table 4.2: State encoding of one feature plane

feature plane	**information**
1	Current hand of the playing agent
2	Union of the other players' hand
3	Recent action one
4	Recent action two
5	History of played cards by player one
6	History of played cards by player two
7	History of played cards by player three
8	Players role (landlord, first/second peasant)
9	Current length of the hand of the players

Table 4.3: State encoding of one feature plane

Another advantage of the RLCard framework is the existence of a strong rule based agent. The first implementations of DQN's in the RLCard paper only managed to win barely 20% as landlord against this rule based agent [41]. Therefore, reaching a higher score against the rule-based-agent can be considered a practical benchmark.

For DouDizhu there is no practical way to evaluate the diversity other than with the diversity losses introduced in chapter 3.1.1. One can not simply check if the actions taken really belong to different strategies, since the transition between playing styles is continuous. They aren't clearly separable such as the decision in the graph game to either turn left or right. Therefore the experiments focus on the evaluation of the diversity losses with respect to the overall score.

The A2C and DDQN are trained as landlord against the rule-based agent. In figure 4.10, the winning rates of both normally trained agents can be seen. The agents reach similar scores with a slight superiority of the A2C.

The analyses of the preference of actions of both agents show that the A2C always tries to play the same actions. It creates a probability estimate of the best actions (like playing a Quad) and gives actions like Pair or Trio low probabilities. It then always tries to play these high value actions, even though most of the time the cards don't allow to play such a combination. The DDQN on the other hand seems to learn which actions are playable but struggles to evaluate

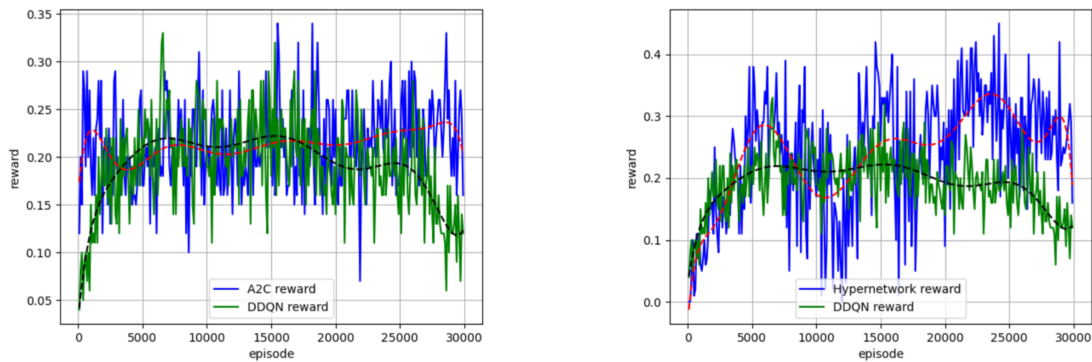


Figure 4.10: Reward of a normally trained DDQN and A2C (left) and the DDQN and the DDQN-hypernetwork without a diversity loss therm (right).

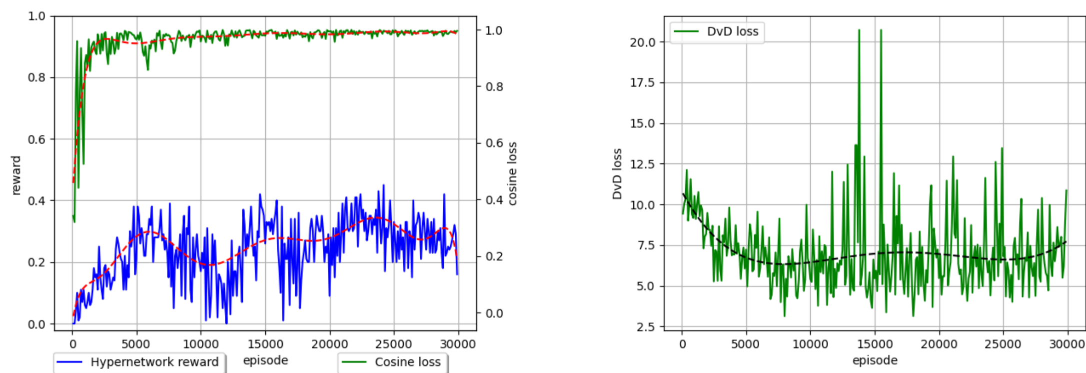


Figure 4.11: Reward and cosine loss (left) and the DvD loss (right) of the hypernetwork trained without a diversity loss.

combinations like the Plane or the Quad correctly. This is probably due to the bias of combination frequency. Since most of the time Solo, Pair and Trio are playable, the DDQN overestimates the values of these combinations. To evaluate diverse playing styles it seems reasonable to evaluate the performance of the DDQN rather than the A2C, because the DDQN agent seems to learn a more realistic strategy instead of always trying to play high card combinations.

In figure 4.10 the performance of the hypernetwork DDQN trained without a diversity term can be seen. The batch size is 2 and the input noise vector size is 4. The learning rate is for all experiments 10^{-5} and the network is trained for 30.000 iterations. It performs better then the vanilla DDQN, even though the score shows higher variance.

In figure 4.11 the cosine loss and the DvD loss are plotted for the hypernetwork trained without a diversity term. While the DvD loss doesn't show any interpretable pattern, the cosine loss shows in the beginning of training still some oscillation, indicating that the hypernetwork produces solutions with at least some amount of diversity, while it still improves its overall performance. At the end, the cosine loss is almost one and the hypernetwork outputs almost

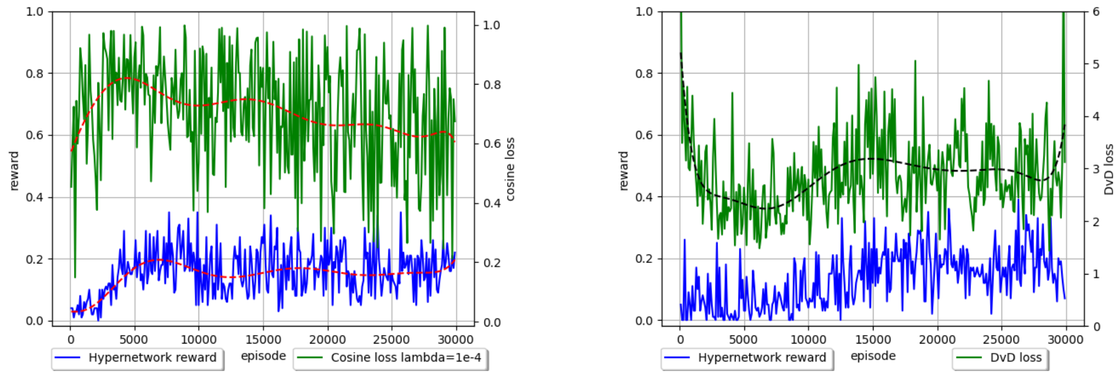


Figure 4.12: Reward and cosine loss (left) and the DvD loss (right) of the hypernetwork trained with a diversity loss term.

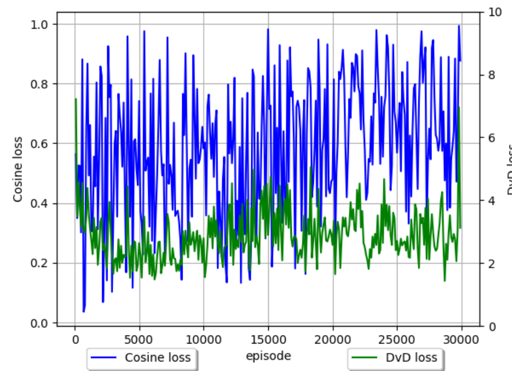


Figure 4.13: Cosine loss of a hypernetwork trained with the DvD loss.

identical agents for arbitrary random vectors as input.

In figure 4.12 the losses of the hypernetwork trained with different diversity terms can be seen. In the beginning of training the cosine loss again shows strong oscillations. They grow stronger as training continues, indicating that the hypernetwork indeed outputs diverse playing agents. This diversity seems to come at the cost of lower performance. While the hypernetwork trained without a diversity loss has a winrate of around 30% after training, the average score of the agents trained with the cosine loss only reach 20%.

The same problem seems to emerge when the hypernetwork is trained with the DvD loss. The performance is highly dependent on the choice of λ, and the diversity term leads to poorer performance of the agents. Additionally, the DvD loss doesn't seem to decrease after 2000 episodes. It doesn't seem obvious, that the hypernetwork outputs diverse agents. To further investigate this issue, in figure 4.13 the cosine loss is plotted for the hypernetwork trained with the DvD loss. The strong oscillations indicate, that the hypernetwork indeed outputs agents with some diversity when compared to the cosine loss for example in figure 4.11. The DvD loss shows a similar effect as the cosine loss quickly after the beginning of training. The strong

diversity at the start might be a reason for the poor performance, especially compared with the cosine loss.

5 Conclusion

The experiments have shown that the hypernetwork approach is applicable to RL-problems. Instead of focusing on the diversity of the generated weights, as has been done in previous work, the objective function can be modified to optimize the hypernetwork with respect to the diversity of the action space of the generated target networks. A variety of agents can be generated with not only high performance in terms of accuracy or score, but also in terms of diverse solution.

The graph game experiments showed that these solutions significantly differ from one another: The agents found two different optimal solutions, albeit the output didn't reach stable solutions like normally trained agents. One reason is that the latent space of the random vectors, which are used as input for the hypernetwork, is not as clearly separable into two distinct domains as the solution space. Therefore, the hypernetwork fails to consistently create weights of agents which belong to one of the two solutions. This might be a general problem of the approach. On the other hand, it could also be due to a variety of not fully understood research questions. No full hyperparameter tuning was performed in this work. Parameters, which are used in the vanilla approaches, might not be optimal for the training with hypernetworks. For example, it is not clear how in the case of DDQN the update rule for the target network should be and if it is reasonable to adopt the rules from the normal approach.

Another question concerns the weight initialization: A great deal of research has been undertaken on the weight initialization of normal neural network. In the course of this work, only the glorot normal initialization led to acceptable results, where as the variance scaling initialization, which was proposed by Deutsch, only worked for extremely simple task like the classification of the MNIST data set. Even for the graph game, this initialization failed to find any solution. Future research should therefore especially focus on better initialization methods.

The loss function of the new hypernetwork approach consists of two parts: A first term which tries to maximize the expected reward, and a second term focusing on the diversity of the generated agents. The two approaches proposed for the latter term, the cosine similarity and the diversity via determinants (DvD), both seem to work in principle. In the graph game experiment, the DvD loss outperformed the cosine loss. A remaining problem is posed by the choice of the scaling parameter λ. The solutions found are highly dependent on this parameter: Chosen too low, the diversity is barely considered during the training of the hypernetwork and no diverse solutions are found. Chosen too high, the hypernetwork fails to maximize the reward and simply finds a set of highly diverse agents. To tackle this problem, Parker-Holder et al. proposed to use a multi-armed bandit approach to switch between the two objectives dynamically. An interesting question for further research in this context would be how the search of diverse, high performing agents should switch between reward maximization and diversity during the course of training.

When compared to population based algorithms like GA and ES, the hypernetwork shows a better performance. It is capable of finding the diverse solutions of the game much faster (6

times the computation time) than the ES. The hypernetwork is therefore able to beat the population based algorithms by mapping random vectors to a diverse solution space. Diverse agents are found more efficiently by combining random noise inputs with a diversity loss function term.

For a more complex game like DouDizhu, the hypernetwork approach fails to find high performing diverse playing agents. While the hypernetwork is able to outperform regularly trained DDQN's when trained without a diversity loss term, it struggles to do so when the objective function is extended by a diversity measure. An interesting research question would therefore the evaluation of other diversity measuring techniques as well as the reformulation or even combination of the ones introduced in this thesis.

Lastly, it was shown that the structuring of the hypernetwork in extractor and weight generator decreased the number of weights of the hypernetwork by a large margin. The remaining network size is still one order of magnitude larger than the main network. Future work on hypernetwork should therefore also focus on other compression techniques. Ideally, the scale of a hypernetwork should be comparable to one of the generated main network.

List of Tables

4.1 Action grouping for DouDizhu . . . 28
4.2 State encoding for DouDizhu . . . 29
4.3 State encoding for DouDizhu . . . 29

1 MNIST hypernetwork architecture: The output of layer2 is divided into vectors of size 15 and fed iteratively to the according weight generators w1, w2, w3, w4. The last weight generator for the output layer is only used once. . . . 43
2 MNIST classifier network architecture . . . 44
3 Graphgame network actor and DDQN: For the DDQN the activation function of the last layer is changed to linear . . . 44
4 Graphgame network critic . . . 44
5 Graphgame hypernetwork architecture (A2C): The first two layers (Extractor) are used to create the embeddings for actor and critic, but different weight generators are used for the distinct networks. . . . 45
6 DouDizu network actor and DDQN: The same architecture is used except for the activation function of the last layer, which is in the case of the DDQN linear instead of softmax . . . 45

7 DouDizu network critic . . . 46
8 DouDizhu hypernetwork architecture (DDQN): The first two layers (Extractor) are used to create the embeddings, the other layers are the weight generators. . . 46
1 Hyperparameters: Image classification . . . 46
2 Hyperparameters: Graphgame . . . 47
3 Hyperparameters: DouDizhu . . . 47

List of Figures

2.1 Tree of reinforcement learning . . . 3
2.2 Gradient clipping in PPO . . . 7
2.3 Pruning hypernetwork . . . 8
2.4 Hypernetwork for one-shot learning . . . 9
2.5 Sampling of plane-3D-shape . . . 9
2.6 Hypernetwork for 3D-shape problem . . . 10

2.7 Diversity of populations 13

3.1 Architecture of the RL-hypernetwork approach 15
3.2 Architecture of the hypernetwork 18

4.1 Training of hypernetwork (image classification) 21
4.2 Path in weight space 22
4.3 Cosine similarity of generated weights 23
4.4 Graph game illustration 23
4.5 Training of A2C and DDQN 24
4.6 Normal and hypernetwork training of DDQN 25
4.7 Hypernetwork diversity loss comparison 25
4.8 Training of hypernetwork with different batch size 26
4.9 Genetic Algorithm and Evolutionary Strategy 27
4.10 A2C, DDQN and hypernetwork DouDizhu 30
4.11 Hypernetwork without diversity loss 30
4.12 Hypernetwork with cosine and DvD loss 31
4.13 Effect of DvD loss on cosine similarity 31

Bibliography

[1] 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.

[2] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[3] Jack Parker-Holder, Aldo Pacchiano, and Krzysztof Choromanski. Effective diversity in population basedreinforcement learning. 2020.

[4] Anuj Mahajan, Tabish Rashid, Mikayel Samvelyan, and Shimon Whiteson. Maven: Multi-agent variational exploration. *arXiv preprint arXiv:1910.07483*, 2019.

[5] Lior Deutsch. Generating neural networks with neural networks. 2018.

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[7] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2013.

[8] Sergey Levine Michael Jordan Pieter Abbeel John Schulman, Philipp Moritz. High-dimensional continuous control using generalized advantage estimation. 2015.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013.

[10] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[11] J. Schulman, M. I. Jordan S. Levine, P. Moritz, and P. Abbeel. Trust region policy optimization. *CoRR, abs/1502.05477*, 2015.

[12] Jane Wang, Zeb Kurth-Nelson, Dharshan Kumaran, Dhruva Tirumala, Hubert Soyer, Joel Leibo, Demis Hassabis, and Matthew Botvinick. Prefrontal cortex as a meta-reinforcement learning system. *Nature Neuroscience*, 21, 06 2018.

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. 2017.

[14] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

[15] David Ha, Andrew Dai, and Quoc V. Le. Multiplicative normalizing flows for variational bayesian neural networks. 2017.

[16] Johannes von Oswald, Christian Henning, Joao Sacramento, and Benjamin F. Grewe. Continual learning with hypernetworks. 2020.

[17] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Tim Kwang-Ting Cheng, and Jian Sun. Metapruning: Meta learning for automatic neural network channel pruning. 2019.

[18] Jack Valmadre Philip H. S. Torr Andrea Vedaldi Luca Bertinetto, Joao F. Henriques. Learning feed-forward one-shot learners. xxxx.

[19] Michael Rotman and Lior Wolf. Electric analog circuit design with hypernetworks and a differential simulator. 2020.

[20] Yoav Chai, Raja Giryes, and Lior Wolf. Supervised and unsupervised learning of parameterized color enhancement. 2019.

[21] Gidi Littwin and Lior Wolf. Deep meta functionals for shape representation. 2019.

[22] Jonathan Lorraine and David Duvenaud. Stochastic hyperparameter optimization through hypernetworks. *https://arxiv.org/pdf/1802.09419.pdf*, 2018.

[23] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

[24] Joseph Suarez. Gan you do the gan gan? 2019.

[25] L. Kozachenko and N. N. Leonenko. Sample estimate of the entropy of a random vector. *arXiv preprint arXiv:1906.07315*, xxx.

[26] D. Podoprikhin D. P. Vetrov T. Garipov, P. Izmailov and A. G. Wilson. Loss surfaces, mode connectivity, and fast ensembling of dnns. 2018.

[27] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. 2018.

[28] Djordje Grbic and Sebastian Risi. Towards continual reinforcement learning through evolutionary meta-learning. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 119–120, 2019.

[29] Whiyoung Jung, Giseung Park, and Youngchul Sung. Population-guided parallel policy search for reinforcement learning, 2020.

[30] Shauharda Khadka, Somdeb Majumdar, Santiago Miret, Stephen McAleer, and Kagan Tumer. Evolutionary reinforcement learning for sample-efficient multiagent coordination. *arXiv preprint arXiv:1906.07315*, 2019.

[31] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. 2018.

[32] David w. Opitz and Jude W. Shavlik. Generating accurate and diverse members of a neural-network ensemble. *arXiv preprint arXiv:1906.07315*, xxx.

[33] Stanislav Fort, Huiyi Hu, and Balaji Lakshminarayanan. Deep ensembles: A loss landscape perspective. *arXiv preprint arXiv:1906.07315*, xxx.

[34] Shashank Singh and Barnabos Poczos. Analysis of k-nearest neighbor distances with application to entropy estimation. *arXiv preprint arXiv:1603.08578*, 2016.

[35] Steijn Kistemaker and Shimon Whiteson. Critical factors in the performance of novelty search. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 965–972, 2011.

[36] Alex Kulesza and Ben Taskar. Determinantal point processes for machine learning. *arXiv preprint arXiv:1207.6083*, 2012.

[37] Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. In *Conference on learning theory*, pages 39–1. JMLR Workshop and Conference Proceedings, 2012.

[38] Osca Chang, Lampros Flokas, and Hod Lipson. Principled weight initialization for hypernetworks. In *International Conference on Learning Representations*, 2019.

[39] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[40] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[41] Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376*, 2019.

Appendices

Appendix 1

Below the architectures of the used networks are noted. Please refer to the Github-link noted above for further information.

layer name	layer components	number of weights	output size
random vector			300
layer1	fully-connected, number of filters: 300 activation: leaky ReLU	90000	300
layer2	fully-connected, number of filters: 855 activation: leaky ReLU	256500	855 (15x57)
w1a	fully-connected, number of filters: 40 activation: leaky ReLU	600	40
w1b	fully-connected, number of filters: 26 activation: leaky RelU	1040	26
w2a	fully-connected, number of filters: 100 activation: leaky ReLU	1500	100
w2b	fully-connected, number of filters: 801 activation: leaky RelU	80100	801
w3a	fully-connected, number of filters: 100 activation: leaky ReLU	1500	100
w3b	fully-connected, number of filters: 785 activation: leaky RelU	78500	785
w4a	fully-connected, number of filters: 60 activation: leaky ReLU	900	60
w4b	fully-connected, number of filters: 90 activation: leaky RelU	5400	90

Table 1: MNIST hypernetwork architecture: The output of layer2 is divided into vectors of size 15 and fed iteratively to the according weight generators w1, w2, w3, w4. The last weight generator for the output layer is only used once.

layer name	**layer components**	**number of weights**	**output size**
input image			28 x 28 x 1
layer1	Convolution: 5 × 5, stride: 1 × 1, padding: 'SAME', number of filters: 32 Activation: ReLU Max pooling: 2 × 2, stride: 2 × 2	832	14 x 14 x 32
layer2	Convolution: 5 × 5, stride: 1 × 1, padding: 'SAME', number of filters: 16 Activation: ReLU Max pooling: 2 × 2, stride: 2 × 2	12816	7 x 7 x 16
layer3	fully-connected, number of filters: 8 activation: ReLU	6280	8
layer4	fully-connected, number of filters: 10 activation: softmax	90	10

Table 2: MNIST classifier network architecture

layer name	**layer components**	**number of weights**	**output size**
input states			16
layer1	Fully-connected, number of filters: 64 activation: ReLU	1088	64
layer2	Fully-connected, number of filters: 64 activation: ReLU	4160	64
layer3	fully-connected, number of filters: 4 activation: softmax	260	4

Table 3: Graphgame network actor and DDQN: For the DDQN the activation function of the last layer is changed to linear

layer name	**layer components**	**number of weights**	**output size**
input states			16
layer1	Fully-connected, number of filters: 64 activation: ReLU	1088	64
layer2	Fully-connected, number of filters: 64 activation: ReLU	4160	64
layer3	fully-connected, number of filters: 1 activation: linear	65	1

Table 4: Graphgame network critic

layer name	layer components	number of weights	output size
random vector			4
layer1 (both)	fully-connected, number of filters: 300 activation: leaky ReLU	1500	300
layer2 (both)	fully-connected, number of filters: 570 activation: leaky ReLU	171570	570 (15x38)
w1c1 (critic)	fully-connected, number of filters: 100 activation: leaky ReLU	1600	40
w1c2 (critic)	fully-connected, number of filters: 136 activation: leaky RelU	13736	26
w1a1 (actor)	fully-connected, number of filters: 100 activation: leaky ReLU	1600	100
w1a2 (actor)	fully-connected, number of filters: 136 activation: leaky RelU	13736	801
w2c1 (critic)	fully-connected, number of filters: 100 activation: leaky ReLU	1600	100
w2c2 (critic)	fully-connected, number of filters: 416 activation: leaky RelU	42016	785
w2a1 (actor)	fully-connected, number of filters: 100 activation: leaky ReLU	1600	60
w2a2 (actor)	fully-connected, number of filters: 416 activation: leaky RelU	42016	90
w3c1 (critic)	fully-connected, number of filters: 100 activation: leaky RelU	1600	90
w3c2 (critic)	fully-connected, number of filters: 260 activation: leaky RelU	26260	90
w3a1 (actor)	fully-connected, number of filters: 100 activation: leaky RelU	1600	90
w3a2 (actor)	fully-connected, number of filters: 65 activation: leaky RelU	6565	90

Table 5: Graphgame hypernetwork architecture (A2C): The first two layers (Extractor) are used to create the embeddings for actor and critic, but different weight generators are used for the distinct networks.

layer name	layer components	number of weights	output size
input states			28 x 28 x 1
layer1	Flatten Layer		30
layer2	Fully-connected, number of filters: 512 activation: ReLU	6280	8
layer3	Fully-connected, number of filters: 512 activation: ReLU	6280	8
layer4	fully-connected, number of filters: 512 activation: softmax	90	10

Table 6: DouDizu network actor and DDQN: The same architecture is used except for the activation function of the last layer, which is in the case of the DDQN linear instead of softmax

layer name	**layer components**	**number of weights**	**output size**
input states			28 x 28 x 1
layer1	Flatten Layer		30
layer2	Fully-connected, number of filters: 512 activation: ReLU	6280	8
layer3	Fully-connected, number of filters: 512 activation: ReLU	6280	8
layer4	fully-connected, number of filters: 512 activation: softmax	90	10

Table 7: DouDizu network critic

layer name	**layer components**	**number of weights**	**output size**
random vector			4
layer1 (both)	fully-connected, number of filters: 300 activation: leaky ReLU	1500	300
layer2 (both)	fully-connected, number of filters: 570 activation: leaky ReLU	6018495	570 (15x38)
w1.1	fully-connected, number of filters: 300 activation: leaky ReLU	4800	300
w1.2	fully-connected, number of filters: 676 activation: leaky RelU	203476	676
w2.1 (actor)	fully-connected, number of filters: 300 activation: leaky ReLU	4800	300
w2.2	fully-connected, number of filters: 513 activation: leaky RelU	154413	513
w3.1	fully-connected, number of filters: 300 activation: leaky ReLU	4800	300
w3.2	fully-connected, number of filters: 513 activation: leaky RelU	154413	513

Table 8: DouDizhu hypernetwork architecture (DDQN): The first two layers (Extractor) are used to create the embeddings, the other layers are the weight generators.

Appendix 2

Below the hyperparameters of the implementations can be found. For further information see the code made available under the link shared above.

Hyperparameter	**Classifier**	**Hypernetwork**
training steps	5000	10.000
learning rate	1e-3	1e-4
decay rate	-	0.99998
weights init	glorot uniform	variance scaling 0.01
batch size	32	32x32
input noise vector	-	300
lamBda	-	1.000, 10.000, 100.000

Table 1: Hyperparameters: Image classification

Hyperparameter	A2C	DDQN	Hypernetwork	Genetic Algorithm
episodes/generations	3000	3000	3000	3000
learning rate	1e-5	1e-5	1e-4	0.01
decay rate	0.9998	0.9998	0.9998	-
discount factor	0.95	0.95	0.95	-
weights init	glorot uniform	glorot uniform	glorot normal	glorot uniform
batch size/ population size	15	15	(2, 8)x15	6
epochs	5	-	-	-
clipping parameter	0.2	-	-	-
entropy coeff.	0.05	-	-	-

Table 2: Hyperparameters: Graphgame

Hyperparameter	A2C	DDQN	Hypernetwork
episodes	30000	30000	30000
learning rate	5e-5	1e-5	1e-5
weights init	glorot uniform	glorot uniform	glorot normal
batch size	32	32	2 x 32

Table 3: Hyperparameters: DouDizhu