Wide and Deep Reinforcement Learning for Grid-based Action Games

Juan M. Montoya and Christian Borgelt
Chair for Bioinformatics and Information Mining, University of Konstanz, Germany

Keywords: Wide and Deep Reinforcement Learning, Wide Deep Q-Networks, Value Function Approximation, Reinforcement Learning Agents.

Abstract: For the last decade Deep Reinforcement Learning has undergone exponential development; however, less has been done to integrate linear methods into it. Our Wide and Deep Reinforcement Learning framework provides a tool that combines linear and non-linear methods into one. For practical implementations, our framework can help integrate expert knowledge while improving the performance of existing Deep Reinforcement Learning algorithms. Our research aims to generate a simple practical framework to extend such algorithms. To test this framework we develop an extension of the popular Deep Q-Networks algorithm, which we name Wide Deep Q-Networks. We analyze its performance compared to Deep Q-Networks and Linear Agents, as well as human players. We apply our new algorithm to Berkley's Pac-Man environment. Our algorithm considerably outperforms Deep Q-Networks' both in terms of learning speed and ultimate performance showing its potential for boosting existing algorithms.

1 INTRODUCTION

In Artificial Intelligence there is an interest in creating rational agents which "act so as to achieve the best outcome or, when there is uncertainty, the best-expected outcome"(Russell and Norvig, 2003, p. 6). The reinforcement learning (RL) problem seeks to develop rational agents that learn from their environment by searching to maximize their outcomes using a rewards system. These RL agents can accomplish different kinds of tasks such as autonomous driving (Kim et al., 2004), playing games (Mnih et al., 2015) and directing robots (Kalashnikov et al., 2018). Since the last decade, RL has been developing exponentially, especially in the area of Deep Reinforcement Learning (DRL) (Henderson et al., 2018).

Some examples of RL agents worthy of mention that have used linear and non-linear functions to improve and extend the RL framework. The autonomous helicopter (Kim et al., 2004) from Standford University is an early work, where the agent learns to hover in place and to fly a number of maneuvers by applying RL via Linear Function Approximation. This implementation is efficient in training, as well as resolves and generalizes the problem of flying and hovering. Nevertheless, it also assumes implicitly that the problem is linearly solvable and so has limited use in many (non-linear) real-world problems. In 2015, Deep Mind's algorithm enabled RL agents to successfully play 49 Atari games using a single algorithm, fixed hyperparameters, and deep learning (Mnih et al., 2015). Most recently, RL agents that control robotic arms learn by applying similar principles how to generalize from their grasping strategies so as to respond dynamically to disturbances and perturbations (Kalashnikov et al., 2018). Those network architectures are robust and able to adapt to many real-world problems; nevertheless, they inherit the already well-known difficulties of choosing and training neural networks and also require a lot of computation power (Goodfellow et al., 2016).

Researchers have been developing and treating Linear Function Approximation and deep learning separately to the best of our knowledge. Why not combine wide learning (e.g. Linear Function Approximation) and deep learning to improve the performance of RL algorithms? Fortunately, a wide and deep machine learning framework has already been developed in the field of recommendation systems (Cheng et al., 2016). Our research aims to develop a framework to transfer this approach to RL, making it easy for researchers to extend already existing DRL algorithms. To test our framework we developed an extension of the popular Deep Q-Networks (DQN) algorithm, which we name Wide Deep Q-Networks (WDQN). We evaluated WDQN using a grid-based

Montoya, J. and Borgelt, C.
Wide and Deep Reinforcement Learning for Grid-based Action Games.
DOI: 10.5220/0007313200500059
In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence (ICAART 2019)*, pages 50-59
ISBN: 978-989-758-350-6

action game: Berkley's Pac-Man environment. We used Berkley's Pac-Man environment because it is highly scalable and computationally efficient. Furthermore, solving the problem of playing Pac-Man is not trivial. The DQN's results on this game are some of the worst among the 49 ATARI games, underperforming humans (Mnih et al., 2015).

Using the simple idea of combining both learning approaches, we demonstrate that our WDQN trained agent has a significantly higher winning rate and produces much better results compared to solely linear or non-linear agents, and has better learning speed compared to DQN.

Our research is now divided into six sections. In the "Background" section we do a review of Linear Function Approximation and DQN and then present our theoretical framework "Wide and Reinforcement Learning", which we used to develop the WDQN algorithm. In the section entitled "Experiments" we show how WDQN performs compared to DQN, Linear Function Approximation, and humans. For this, we expose how Berkley's Pac-Man environment works, the experimental set-up for WDQN, and the results. Next, we discuss the results and then present our conclusions.

2 BACKGROUND

2.1 Linear Function Approximation

RL agents receive feedback from their actions in the form of rewards from interacting with the environment. The agents aim to resolve a sequential decision problem by optimizing the cumulative future rewards (Sutton and Barto, 2018). One of the most popular methods to resolve this is Q-learning (Watkins, 1989; Henderson et al., 2018). Nevertheless, Q-Learning alone cannot compute all value functions when confronted with a large state space, which is the case for most real-world problems (Russell and Norvig, 2003).

One way of tackling the large state space problem is to use a function $\hat{Q}$ to approximate the true q-value Q. Differentiable methods, such as linear combination of features and neural networks, offer us the possibility of using stochastic gradient descent (SGD) as an intuitive form to optimize the action value function.

The equation using a Q-learning update after taking action A_t in state S_t observing the immediate rewards R_{t+1} and continuing state S_{t+1} is then

$$\theta_{t+1} = \theta_t + \alpha(y_t^{\hat{q}} - \hat{Q}(S_t, A_t; \theta_t)) \bigtriangledown_{\theta_t} \hat{Q}(S_t, A_t; \theta_t) \quad (1)$$

where α is a scalar step size, θ_t the parameters of the function $\hat{Q}$, and the target function $y_t^{\hat{q}}$ defined as $R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a; \theta_t)$. Gradient descent is applied by optimizing a loss function from the difference of $y_t^{\hat{q}}$ and $\hat{Q}(S_t, A_t; \theta_t)$.

The value function can be approached using a linear combination of features $f(S_t) = [f_1(S_t), ..., f_n(S_t)]^t$. Each $f_i(S_t)$ represents a feature mapped at state S_t with a particular function f_i. The q-value function is constructed as $Q^{LN}(S_t, A_t; \theta_t^{LN}) = f(S_t)^T \theta_t^{LN}$, where θ_t^{LN} are the weights of the linear function. The target is then defined as

$$y_t^{LN} := R_{t+1} + \gamma \max_a \hat{Q}^{LN}(S_{t+1}, a; \hat{\theta}_t^{LN}) \quad (2)$$

The differential of (1) applied to $f(S_t)^T \theta_t^{LN}$ is then $\theta_{t+1}^{LN} = \theta_t^{LN} + \alpha(y_t^{LN} - Q^{LN}(S_t, A_t; \theta_t^{LN}) f(S_t)$. It is therefore a simple matter to compute the update rule.

In practice, linear methods can be very efficient in terms of both data and computation. Nevertheless, prior domain knowledge is usually needed to create useful features, representing interactions between features can be difficult, and convergence guarantees are limited to linear problems (Sutton and Barto, 2018).

2.2 Deep Q-Networks

The intuitive action to resolve non-linear cases is to substitute the approximation function $\hat{Q}$ with a non-linear function using neural networks. However, this first "naive" approach underperformed because of problems with non-stationary, non-independent, and non-identically distributed data (Mnih et al., 2015).

The DQN tackle such problems by using an experience replay memory and target networks. DQN use a convolutional neural network (convNets) architecture to compute the state S_t to a vector of action values. This q-value function is $Q^{DQN}(S_t, A_t; \theta^{DQN})$, where θ^{DQN} are the parameters of the convNets. The experience replay (Lin, 1992) saves observed transitions for some time in a dequeue. These transitions are later uniformly sampled and used to update θ^{DQN}. The parameters $\hat{\theta}^{DQN}$ of the target network $\hat{Q}^{DQN}$ are copied from the online network every τ steps, so that $\hat{\theta}^{DQN} = \theta^{DQN}$, fixing $\hat{\theta}^{DQN}$ on all other steps. The target used by DQN is then

$$y_t^{DQN} := R_{t+1} + \gamma \max_a \hat{Q}^{DQN}(S_{t+1}, a; \hat{\theta}^{DQN}) \quad (3)$$

Both components dramatically improve the performance of the algorithm (Mnih et al., 2015) and have been successfully extended since their creation (Henderson et al., 2018). However, DQN and variants inherit all the problems related to neural networks such as the difficulty of interpreting the decision making

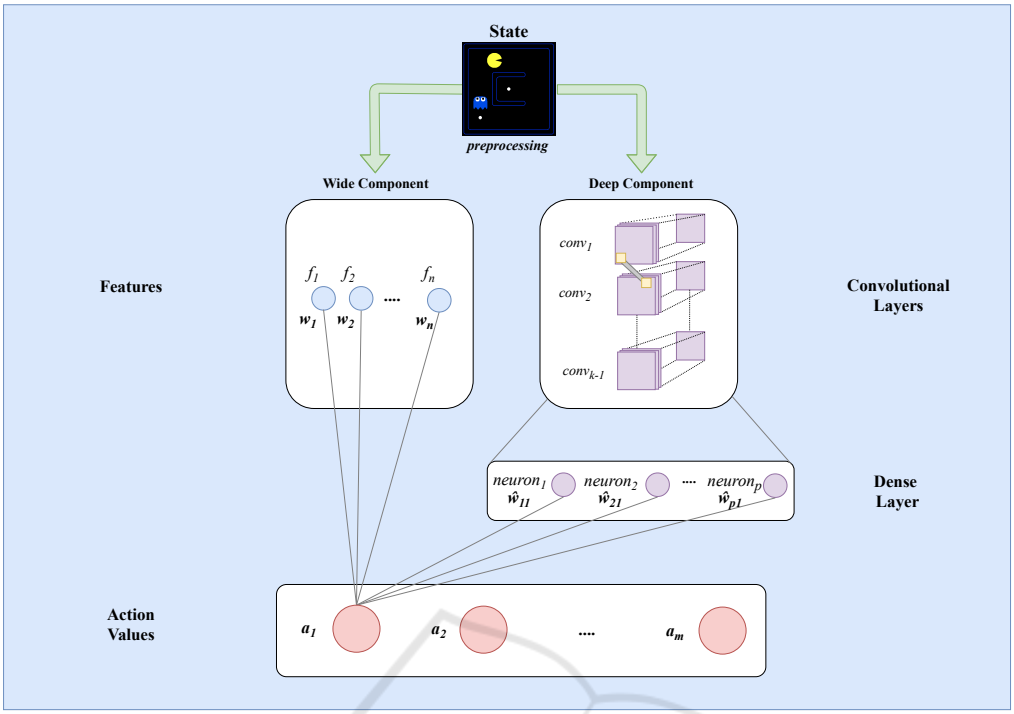


Figure 1: Wide and Deep Reinforcement Framework showing the connections from the weights of the deep and wide component to action value a_1.

of the networks, the tuning of multiple hyperparameters (Goodfellow et al., 2016), and the complexity in computation with no guarantees of convergence (Sutton and Barto, 2018).

Our approach proposes to combine both linear and non-linear methods in order to obtain better, faster and more comprehensive results using DQN Algorithms. Cheng et al. (2016) already showed that such a wide deep learning model is viable and significantly improves the results for recommendation systems.

3 WIDE AND DEEP REINFORCEMENT LEARNING

Figure 1 shows the general structure of the Wide and Deep Reinforcement Learning (WDRL) Framework, which can be used for already existing DRL algorithms. This framework consists of the linear combination of features (left side) and convNets (right side), which are respectively called the wide and deep component. Both components compute the action values $A = [a_1, ..., a_m]$.

Initially, the states are preprocessed separately for the wide and deep component. For this step, the preprocessing function ϕ needs to be able to process the state S_t separately for the wide and the deep component. The state S_t needs to be processed independently for each component because the inputs are different for each component. The wide component requires information provided by the features, while the deep component takes an image representation to be processed by the convNets in our case.

3.1 The Wide and Deep Component

In Figure 1, the wide component uses a linear combination of features $f_1, ..., f_n$, which are connected with their respectively weights $\theta^{LN} = [w_1, ..., w_n]^t$ to each value action a_i of the approximation functions. In this illustration, we represent the weights as shared weights (e.g. as a vector) because they correspond to the implementation used in our experiments. The features can be represented by a feature matrix $F \in \mathbb{R}^{m,n}$, where m is the total number of outputs and n the number of features. Figure 1 shows $[F^{1,1}, ..., F^{1,n}] = F^{1n}$ multiplied by θ^{LN} for a_1. The optimization step can be easily inferred from (1).

In our illustration, the deep component uses the convNets $[conv_1, conv_2, ..., conv_{k-1}]$ for almost all layers. Each hidden layer except for the last one performs the following computation $a^{(l+1)} = \hat{f}(\hat{W}^{(l)}a^{(l)} + b^{(l)})$, where l is the layer number and $\hat{f}$ is the activation function, $a^{(l)}$, $b^{(l)}$, and $\hat{W}^{(l)}$ are the activations, bias, and function weights at the l-th layer. The last layer k implements a dense layer,

but it could also be extended to have more fully connected layers. In addition, this final layer does not use an activation function so that $a^{(k)} = \hat{W} a^{(k-1)} + b^{(k-1)}$, where $\hat{W}^{(k-1)} = \hat{W} \in \mathbb{R}^{pm}$. Figure 1 shows also that $a^{(k-1)} = [neuron_1, ..., neuron_p]^t$ and the weights of the last layer for a_1 are represented by $[\hat{w}_{11}, ..., \hat{w}_{p1}]^t = \hat{W}^{p1}$.

The wide and deep components are combined to generate the output of the Wide and Deep Reinforcement function. Figure 1 shows that $a_1 = F^{1,1} \cdot w_1 + ... + F^{1,n} \cdot w_n + neuron_1 \cdot \hat{W}^{1,1} + neuron_1 \cdot \hat{W}^{1,1} + ... + neuron_p \cdot \hat{W}^{p,1}$, which is equal to $a_1 = F^{1n} \cdot \theta^{LN} + a^{(f-1)} \cdot \hat{W}^{p1}$. This can be easy generalized to all action values: $A = F \cdot \theta^{LN} + a^{(k)}$.

Now, in order to train the function, two approaches can be implemented. The first approach is called joint ensemble training and computes SGD jointly for the linear and non-linear functions. We call the second approach *semi*-ensemble training because, in contrast to ensemble training in supervised machine learning, predictions of the linear and non-linear functions do influence each other during training (Cheng et al., 2016). However, as in ensemble training, this approach implements SGD separately for both functions. Both approaches use the combined prediction of the linear and non-linear function to act.

3.2 Wide Deep Q-Networks

The DQN Algorithms can be extended by integrating the linear function Q^{LN} with the non-linear function Q^{DQN} creating the combined function Q^{WD}; thus called "Wide Deep Q-Networks".

For our illustration using WDQN, the wide component uses the target function for the linear combination of features shown in (2). Meanwhile, the deep component uses the target function of (3). Therefore, the combined function is $Q^{WD}(S_t, A_t; \theta^{WD}) = Q^{LN}(S_t, A_t; \theta^{LN}) + Q^{DQN}(S_t, A_t; \theta^{DQN})$, where θ^{WD} includes the parameters of the wide and deep function.

For the joint training, the algorithm remains almost identical to the original DQN. The online Q^{DQN} and target network $\hat{Q}^{DQN}$ need only be replaced by Q^{WD} and $\hat{Q}^{WD}$ respectively. The target is then defined as

$$y_t^{WD} := R_{t+1} + \gamma \max_a \hat{Q}^{WD}(S_{t+1}, a; \hat{\theta}^{WD}) \quad (4)$$

where $\hat{\theta}^{WD}$ are the target parameters of the combined function. The SGD is estimated directly on the joint function.

For the semi-ensemble training, the algorithm needs to save the linear and non-linear function, as well as the combined function (see Algorithm 1). Basically, the actions are being chosen by the combined function Q^{WD}, however SGDs are estimated separately on Q^{LN} and Q^{DQN}, implementing both targets from (2) and (3).

Algorithm 1: Semi-Ensemble Training WDQN.

Initialize:replay memory D to size N;
Action-value functions Q^{WD}, Q^{LN}, Q^{DQN} with respectively random weights θ^{WD}, θ^{LN}, θ^{DQN};
Target action-value functions $\hat{Q}^{WD}$, $\hat{Q}^{LN}$, $\hat{Q}^{DQN}$ with weights $\hat{\theta}^{WD} = \theta^{WD}$, $\hat{\theta}^{LN} = \theta^{LN}$, $\hat{\theta}^{DQN} = \theta^{DQN}$ respectively.
for episode = 1, M do
  Initialize sequence $S_1 = [x_1]$
  Preprocessed sequence $\phi_1 = \phi(s_1)$
  for t = 1,T do
    With probability ε select a random action $a_t \in A_t$ otherwise select: $a_t = argmax_a Q^{WD}(\phi(s_1), a; \theta^{WD})$
    Execute action a_t, observe reward R_t and image x_{t+1}
    Set $S_{t+1} = S_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(S_{t+1})$
    Store transition $(\phi_t, a_t, R_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_t, a_t, R_t, \phi_{t+1})$ from D
    Set y_j^{DQN}, $y^{LN} = r_j$ for terminal $\phi(S_{j+1})$ and non terminal $\phi(S_{j+1})$: $y_j^{DQN} = R_{t+1} + \gamma \max_a \hat{Q}^{DQN}(S_{t+1}, a; \hat{\theta}^{WD})$ $y_j^{LN} = R_{t+1} + \gamma \max_a \hat{Q}^{LN}(S_{t+1}, a; \hat{\theta}^{LN})$
    Perform gradient descent on $(y_j^{DQN} - Q(S_{t+1}, a; \theta^{DQN}))^2$ and $(y_j^{LN} - Q(S_{t+1}, a; \theta^{LN}))^2$ with respect to θ^{LN} and θ^{DQN}
    Every C steps reset $\hat{\theta}^{LN} = \theta^{LN}$, $\hat{\theta}^{WD} = \theta^{WD}$ and $\hat{\theta}^{DQN} = \theta^{DQN}$.
  end for
end for

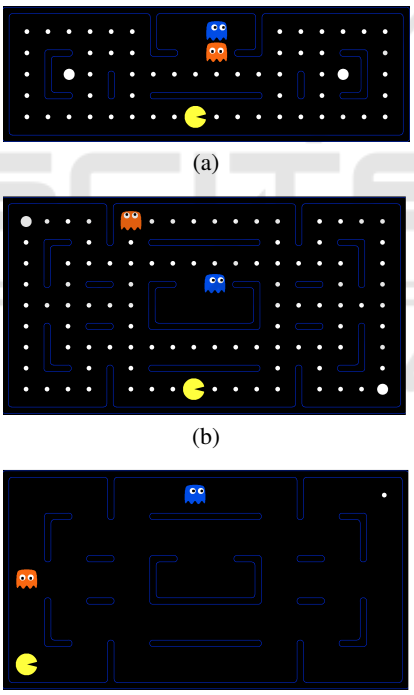
4 EXPERIMENTS

4.1 The Pac-Man Environment

In order to compare the different algorithms with each other, we used the Pac-Man open source environment of UC Berkeley (DeNero and Klein, 2010). Our goal was not to use a fully realistic simulator of Pac-Man to achieve superhuman results, but rather to have a scalable and computer efficient environment to test our Deep and Wide Reinforcement framework. The Pac-Man environment of UC Berkeley is suitable for this: the scalability is guaranteed by providing customizable map sizes. Moreover, the preprocessing of the game states is more efficient than using raw pixels

(more details coming below). We decided to test our approach only on small and medium maps due to our computational limitation of one 12 GB NVIDIA Titan GPU. We analyzed the performance of our agents for each map. We have chosen ConvNets because they share weights across maze positions creating independence between maze locations and accelerating training speed. This is an advantage compared to fully connected nets with the same amount of layers (Goodfellow et al., 2016), reassuringly, the use of ConvNets has been a standard tool for DRL research (Mnih et al., 2015; van Hasselt et al., 2016; Henderson et al., 2018).

For a more complex and fully realistic Pac-Man environment see Ms. Pacman used in (Mnih et al., 2015; van Hasselt et al., 2016). In order to guarantee transparency and reproducibility, our Python code using Tensorflow is available at GitHub[1]. We follow the recommendations of Henderson et al. (2018) to include the used hyperparameters and random seeds.



(a)

(b)

(c)

Figure 2: The small (a) and medium (b) maps of Berkeley's Pac-Man environment at the agent's starting point. The medium map in (c) shows the last dot that has to be eaten in order to finish the game.

Figure 2 shows our implemented maps in Berkeley's Pac-Man environment. Pac-Man, the yellow agent, can move vertically and horizontally, eating dots on his way. The goal of Pac-Man is to score as many points as possible by eating the small dots all around the maze while avoiding crashing into any of the ghosts that are chasing him. Additional points are given if Pac-man eats the special edible ghosts. The ghost becomes temporally edible when Pac-Man eats one of the big dots on the board. Eaten ghosts reappear back as not edible ghosts. An episode is finished when Pac-Man either eats the last dot (see (c) in Figure 2) or gets killed by a non-edible ghost.

In both maps, Pac-Man starts in the top middle part of the map which also contains two ghosts, many small dots, and two big dots (see (a) and (b) in Figure 2). The room, where the ghosts start and reappear if they get killed, is in the middle of the maze for medium map and at the top of the maze for the small map. For scoring, we used the original reward system of UC Berkeley. The initial score is always zero and restarts after each episode. Eating the small and big dots scores 10 points. For each eaten edible ghosts, the agent scores 50 points. In order to avoid stagnation, the agent is deducted 1 point for each second that is spent. At the end of an episode, Pac-Man either wins, scoring 100 points or loses, deducting 500 points.

A profitable approach to achieve higher scores is to eat ghosts because of the high rewards. Nevertheless, the original DQN algorithm did not learn to eat them (Mnih et al., 2015), explaining to some extent the poor scores of DQN in this game. Their DQN agent received a clipped reward of either 1 or -1 at each state S_t. For example, the agent gets 1 point for eating either a ghost or a dot; a negative score, for dying, is -1. The agent, therefore, did not learn the significance of eating ghosts because of the low reward (van Hasselt et al., 2016). However, the agent did learn to eat ghosts in two other approaches: Hasselt et al. (2016) confronted this problem by adaptively normalizing the targets of the network, making it possible to process all types of rewards. Meanwhile, Van der Ouderaa (2016) used the incoming reward of Berkeley's environment at each state S_t to train the DQN algorithm.

For our approach, we preferred to use Van der Ouderaa's method, because it keeps our implementation as minimalistic as possible and was also tested using the same reward system. In addition, in our experiments, we decided to distinguish whether the trained agents can or cannot eat ghosts. We consider that an agent can only eat a ghost when it actively hunts the edible ghost and not purely by chance. To do this, we observed at least 10 games for each of the selected agents.

In order to save computational power, raw pixels are not used as the input for the deep component (also

[1]https://github.com/JuanMMontoya/WDRL

for DQN). For agents that learn directly from pixels see (Mnih et al., 2015; van Hasselt et al., 2016). The input we implemented, consists of an array with six matrices containing the coordinates respectively of each 1) ghost, 2) wall, 3) dot, 4) big dot, 5) Pac-Man and 6) edible ghost (van der Ouderaa, 2016). In each matrix, a 0 or 1 respectively expresses the existence or absence of the element at each coordinate on its corresponding matrix. These matrices are easily retrievable for each state and they are a distinctive quality of using Berkeley's environment.

The matrix size is defined by the width W and height H of the game grid. The input, therefore, has the following dimensions of $W \times H \times 6$. This permits a fast identification of the important game elements. In combination with the size favorable maps, the inputs' preprocessing and the back-propagation are computed efficiently.

For the wide component, we used the Linear Function Approximation contained in the Berkeley environment. This consist of the three features structured in the following way:

1. ***#-of-ghosts-1-step-away***: lets the agent know the number of ghosts one step away and does not differentiate between edible and non-edible ghosts.
2. ***eats-food***: sets to one if there is a ghost one step away and zero if there isn't.
3. ***closest-food***: gives the direction to the closest dot.

Notably, the linear agent cannot eat ghosts. We found that the main reason for such a behavior is the inability of *#-of-ghosts-1-step-away* to distinguish the ghost type. This creates a dichotomy: either learning to eat ghost or avoiding them. Since the reward incentives are higher to survive he chooses to evade them.

4.2 Experimental Set-up

The algorithms analyzed are the Linear Function Approximation, DQN, and WDQN. To tune the hyperparameters we performed around 100 preliminary experiments for the linear and DQN agent in different maps. We adjusted mainly the size of the memory replay, the learning rates, the update rate of the target function, the network structure, and the exploration value ε with its final exploration frame. We consistently maintained the final hyperparameters shared between the linear and DQN agent. Afterward, we used these hyperparameters for the WDQN. All agents also have also the same action values, i.e. four possible directions: left, right, up, down.

In addition, we found that the learning curve stagnated around 10000 episodes. Therefore, we have chosen this value as the training limit for the final experiments. In order to compare the agent's performance against each other, we decided to use the averaged score and the win rate of each agent for 100 episodes. Finally, we repeated multiple random seeded experiments with the same hyperparameters for each selected agent to guarantee consistency.

Our DQN-agent do not apply the same hyperparameters but rather the same algorithm structure described in (Mnih et al., 2015). Our convNet has two convolutional layers and one fully connected layer that maps into the four outputs. The first layer applies 16 3×3 filters with full padding and one stride, while the second 32 3×3 filters with full padding and one stride. The fully connected layer has 256 neurons. The learning rate was set up to 0.001 using ADAM optimization algorithm. This architecture permits us to maintain the network small but with the capacity of making complex decisions. By using height and width 3x3 filters in two layers (resulting in a 5x5 field of view) it permits the agent to see at least 2 steps away from him. This is important for avoiding been eaten. The two layers with a depth of 16 and 32 dimensions respectively allow the agent to be able to abstract from a combination of 32 different base maze patterns. A similar architecture was implemented in (van der Ouderaa, 2016) and was confirmed during our preliminary experiments.

Using the linear combination of features, the linear agent chooses the policy at state s_n that could give him the highest q-function at the next state s_{n+1}. The sequence of training for our linear approximator follows the DQN's algorithmic structure (i.e. target function, memory replay, etc.). The learning rate was set up to 0.1 using SGD. The exact description of the applied hyperparameters for all agents can be found in our GitHub's repository.

The WDQN algorithm is trained using semi-ensemble training because of the preliminary knowledge of learning rates for the linear function and convNets. The listed features of the last subsection are implemented in the wide component using different combinations of features. The WDQN algorithm is tested separately using a wide component with three, two, and one feature(s) respectively. We decided to combine the features in the following way because:

1. Combining the three available features permits us to mix DQN and the Linear Approximator completely.
2. By avoiding using the feature *#-of-ghosts-1-step-away*, the two features *closest-food* and *eats food* do not contain the strongest constraint to not learn eating ghosts.
3. The feature *#-of-ghosts-1-step-away* should enable faster learning to win, because it gives the

most important information on how to survive the game.

Table 1: Score Average and Win rate in Small and Medium Map, as well as whether the agents learned to eat ghosts. The agents presented are the Linear Function Approximation, DQN and WDQN with 3, 2 and 1 features respectively, as well as the Random agent. For each algorithm, the best agent is chosen and evaluated for 100 episodes. The best amateur human player also played 100 games.

	Small Map		Medium Map		Eats
	Score	Win Rate	Score	Win Rate	Ghosts
Linear	-108	18%	486	55%	no
DQN	110	33%	622	47%	yes
WDQN 3 feat.	296	60%	666	64%	no
WDQN 2 feat.	**353**	**61%**	**727**	**65%**	**yes**
WDQN 1 feat.	215	51%	614	61%	no
Human	-99	11%	125	12%	yes
Random	-463	0%	-443	0%	no

4.3 Results

Figure 3 shows the achieved scores for each agent in the small (right side) and medium map (left side) during training. A clear difference of training speeds is observable between algorithms. The WDQN 3 and 2 features agents learn faster than the WDQN 1 feature and the DQN agents. The linear agent learns faster than all other agents but stabilizes once arriving at a certain threshold, which is exceeded by all agents using neural networks at some point. The learning speed also varies depending on the map.

In the medium map, we see that the training is faster than in the small map. For instance, in the medium map, the WDQN 3 and 2 features surpass the score of the linear agent at 4000 episodes and DQN at 6700 episodes. This results in a difference of 2700 episodes between WDQN and DQN. Meanwhile, the WDQN 3 and 2 features agents in the small map reach the score of the linear agent at 3900 episodes, while DQN at 5700 episodes. This is a difference of 1600 episodes. Thus, there is a substantial disparity of training speed between WDQN and DQN agents, which also varies depending on the map.

In addition, Figure 3 presents the different performance of the algorithms during training. WDQN 3 and 2 features perform better than the WDQN 1 feature and DQN agents. The linear agent achieves the worst results followed by WDQN 1 feature, while DQN stays only behind the outcomes of WDQN 3 and 2 features. Yet again, there are notable differences between maps. First, the WDQN 3 features agent behaves better in the small map than in the medium map. Second, the performance gap between worst and better agent is more unequivocal in the small than in the medium map.

Table 1 shows the averaged score and winning rate for WDQN, DQN, Random and Linear agents, and human players in the small and medium map for 100 episodes. Table 1 illustrates that the WDQN 3 and 2 features have the best averaged score and win rate among all agents, while the DQN agent is at 3^{rd} or 4^{th} place depending on the map. The best agent is WDQN 2 features. The agent wins 61% with 353 points and 65% with 727 points respectively for the small and medium map and learns to eat ghosts. At the same time, the WDQN 3 features agent has a slightly smaller win rate than WDQN 2 features but scores lower than such an agent and cannot eat ghosts.

The DQN agent underperforms those agents and learns to eat ghosts as shown in Table 1. However, in the medium map, the difference in score to WDQN 3 and 2 features is less than 120 points, although the DQN agent loses more than half of the games. In contrast, the linear agent wins a little more than half of the games but scores around 120 points less than DQN. As already stated, the linear agent cannot eat ghosts because of the features being used. The WDQN 1 feature agent wins less than the other WDQN agents and cannot eat ghosts. In the case of the medium map, it scores lower than the DQN agent.

Furthermore, the results for our amateur human players are below the non-random agents with around 11% of games won. The only slight exception is the score (-99) against the linear agent in the small map (-108). For the human row, we selected the best players of a round-robin tournament with 9 volunteers (see our GitHub repository for the recorded human's games). This shows us that the problem is not trivial for humans. Random agents have the worst results (-463 and -443); thus, moving by chance is not a profitable option for this game.

The Figure 3 and Table 1 present almost equivalent outcome. Yet, the most notable contrast is that the best agent during training is WDQN 3 features in Figure 3 (left side), but the best scores in Table 1 are produced by WDQN 2 features.

The present findings confirm that the WDQN agent can perform better than DQN, linear and random agent, as well as the amateur human players. WDQN algorithm with 3 and 2 features have the best score and win rates. In addition, they learn faster than the DQN agent. Nevertheless, the WDQN 1 feature agent has a lower score and learning speed than the other WDQN's. Our assumption that WDQN 1 feature could learn faster is proven to be wrong.

Lastly, we detect some tendencies in the results that are worth examining more closely. There is a learning speed difference between maps that could be related to the performance of the linear agent. More-

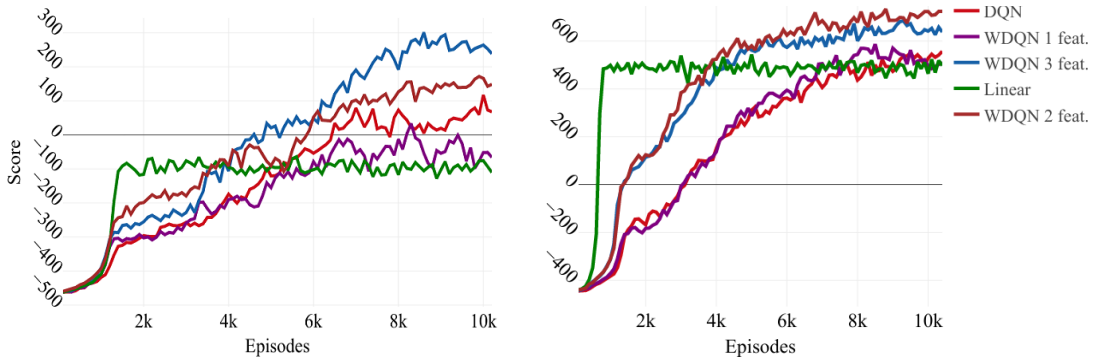


Figure 3: Each agents score is evaluated during training for the small (right side) and medium map (left side). Five random seeded training sessions are conducted for each agent with the same hyperparameters until 10200 episodes respectively in the small (a) and medium map(b). The lines arise of averaging those training sessions according to the agent type.

over, there are algorithms that learn to eat ghosts, while others do not, depending on the implemented feature. Finally, making only random moves is a high unproductive strategy both to win and score high. This could be part of a pattern that explains the negative results of the DQN agent.

5 DISCUSSION

The results presented confirm the better performance of the combined WDQN agent compared to the solely DQN and linear agent. Now, we concentrate on discussing the reasons for such improved performance by analyzing the learning speed difference between maps, why some agents eat ghosts or not, and the logic behind the DQN's underperforming.

For WDQN 2 and 3 features agents, we observe a speedup of the learning during training results, which depends on the map. The linear agent's performance can be the key to understand such a difference. In the medium map, the linear agent works considerably better than on the small map. The superior results of the linear agent in the medium map influence those WDQN agents to learn faster and more prolonged than in the small map.

In addition, we observe that the poor results of the linear agent are not transferred to the WDQN agents in the small map. This could imply that our linear function cannot abstract a proper solution to the game, although the features itself provide valuable information.

The scores and win rates of the trained agents show that the WDQN agents with the *#-of-ghosts-1-step-away* feature do not learn to eat ghosts. This explain why the WDQN 2 features outperforms the WDQN 3 features learns to eat ghosts scoring more. Furthermore, by looking at WDQN 1 feature's performance, the *#-of-ghosts-1-step-away* feature helps to develop the capacity to survive in short-term. Yet, it restricts the capacity to achieve high scores in long term.

We observed that the weights of the wide component change faster and stronger than those of the deep component because of its simple updates. In the case of WDQN 3 feature, there is an information conflict between the deep component and the wide component. The features of the linear component treat all ghosts identically, while the input of deep component can see the difference between edible and not. From this conflict the wide component is more dominant because of its weights. If there is no compensation of this effect, for example, by normalizing the weights, switching on and off the wide component during training or only training more episodes, we suspect that WDQN agents using this feature will not learn to eat ghosts.

Repetitive observations of the DQN agent's play leads us to detect that the agent had difficulties reaching the last dots in the map. This was especially true if the dots were far away from the agent. Figure 2 (c) illustrates exactly such a case, where the agent is considerably far for the last dot. We believe that:

1. Reaching the last dot is complicated because the reward propagation usually happens in a different place each game.
2. Making random moves does not permit Pac-Man to explore the map because the ghosts can easily kill him when maneuvering randomly. Rather, the DQN agent seems to reach far away dots thanks to the movements caused by avoiding ghosts.

In both cases, there are insufficient examples to learn how to detect the exact position of far away dots. This could explain the success of the WDQN 3 and 2 features. The wide component adds the information about where to find such dots.

However, we should not exclude the possibility that the DQN problems could be related to the chosen hyperparameters, especially of the convNets. Maybe choosing larger filters could contract such problem. Moreover, assuming that the DQN has an excellent solution in itself in another different implementation, maybe adding a wide component would improve neither the training speed nor the results.

Conclusively, we believe that integrating a good wide component to the WDQN model can be the reason for a substantial speedup of learning. Adding a deep component to a linear agent could improve its linear limitation considerably by converting it into a non-linear model. Precautions are needed when choosing which features to integrate into the combined agent. Lastly, a favorable wide component can compensate for the difficulties of the deep component to learn from insufficient examples.

6 CONCLUSION

Our research shows that the WDQN agents can outperform linear and DQN agents in score, winning rate and learning speed. The chosen features also play a role in achieving these results. However, there can be learning limitations depending on the selected feature(s). The research demonstrates that combining a neural network with a linear agent helps improve results by allowing the model to learn non-linear relationships while adding information about the interaction between specific features, while also making the agent adaptable to uncertainty. Furthermore, the wide component can complement the weaknesses of a non-linear agent by helping the agent learn faster and concentrate on finding less obvious important features.

Our method is straightforward and employable for various deep reinforcement contexts. For real-world implementations such as robotics, the combination of linear and non-linear functions in our Wide and Deep Reinforcement Learning provides an interesting tool for integrating new devices like sensors in the form of features into DRL agents, or for including better expert knowledge with human chosen features. Future work could look into extending WDQN to include newer DQN-related algorithms and developing methods that make implementing WDQNs easier; for example, automatically setting the learning rate of the wide component from the deep component's to reduce the number of hyperparameters. In addition, one could research how to ensure that the influence of negative features of the wide component can be overridden by the deep component.

ACKNOWLEDGEMENTS

We thank our colleagues from the Chair for Bioinformatics and Information Mining of the University of Konstanz, who provided insight and corrections that greatly assisted the research. Especially, we are grateful to Christoph Doell and Benjamin Koger for the continuous assistance with the paper's structure and the engineering of the deep neural networks.

REFERENCES

Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X., and Shah, H. (2016). Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, pages 7–10, New York, NY, USA. ACM.

DeNero, J. and Klein, D. (2010). Teaching Introductory Artificial Intelligence with Pac-Man. *Proceedings of the Symposium on Educational Advances in Artificial Intelligence*, pages 1885–1889.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018). Deep Reinforcement Learning that Matters. In *Proceedings of the Thirtieth-Second AAAI Conference on Artificial Intelligence*, AAAI'18. AAAI Press.

Kalashnikov, D., Irpan, A., Pastor, P., Ibarz, J., Herzog, A., Jang, E., Quillen, D., Holly, E., Kalakrishnan, M., Vanhoucke, V., and Levine, S. (2018). QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic. *CoRR*, abs/1806.10293.

Kim, H. J., Jordan, M. I., Sastry, S., and Ng, A. Y. (2004). Autonomous Helicopter Flight via Reinforcement Learning. In Thrun, S., Saul, L. K., and Schölkopf, B., editors, *Advances in Neural Information Processing Systems 16*, pages 799–806. MIT Press.

Lin, L.-J. (1992). Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning*, 8(3):293–321.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-Level Control through Deep Reinforcement Learning. *Nature*, 518(7540):529–533.

Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education, 3 edition.

Sutton, R. S. and Barto, A. G. (2018). Introduction to Reinforcement Learning. Working Second Edition.

van der Ouderaa, T. (2016). Deep Reinforcement Learning in Pac-Man. Bachelor Thesis, University of Amsterdam.

van Hasselt, H. P., Guez, A., Hessel, M., Mnih, V., and Silver, D. (2016). Learning values across many orders of magnitude. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4287–4295.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK.