
PairVDN - Pair-wise Decomposed Value Functions

Zak Buzzard

Department of Computer Science
University of Cambridge
zzb20@cam.ac.uk

Abstract

Extending deep Q-learning to cooperative multi-agent settings is challenging due to the exponential growth of the joint action space, the non-stationary environment, and the credit assignment problem. Value decomposition allows deep Q-learning to be applied at the joint agent level, at the cost of reduced expressivity. Building on past work in this direction, our paper proposes PairVDN, a novel method for decomposing the value function into a collection of pairwise, rather than per-agent, functions, improving expressivity at the cost of requiring a more complex (but still efficient) dynamic programming maximisation algorithm. Our method enables the representation of value functions which cannot be expressed as a monotonic combination of per-agent functions, unlike past approaches such as VDN and QMIX. We implement a novel many-agent cooperative environment, Box Jump, and demonstrate improved performance over these baselines in this setting. We open-source our code and environment at <https://github.com/zzbuzzard/PairVDN>.

1 Introduction

Multi-agent reinforcement learning (MARL) deals with the application of reinforcement learning to settings with more than one agent, for instance team-based videogames, the coordination of multiple self-driving vehicles, or competitive games such as chess or go. However, the complexity of multiple agents brings various challenges for reinforcement learning: the size of the joint action space grows exponentially with the number of agents, quickly becoming intractable to enumerate; credit assignment may become difficult, particularly in environments with joint reward systems; the presence of other agents creates a non-stationary environment from the perspective of individual agents, who must now account for the behaviour of the other agents (which changes during optimisation). This paper examines the fully cooperative multi-agent setting, in which several agents jointly work for the same goal, with a single reward signal shared by all agents.

Deep Q-networks (DQN) Mnih et al. [2015] are a method for learning policies via deep learning, and has seen success across a range of tasks. However, as DQN requires maximisation over the entire joint action space, this is not feasible in a multi-agent setting with more than a few agents. While it is possible to apply DQN individually to each agent - a naive approach called independent Q-learning (IQL) - the non-stationarity of the environment in this setting causes low performance Rashid et al. [2020]. To overcome these issues, Value Decomposition Networks (VDN) Sunehag et al. [2018] were proposed, which avoid the high cost of iterating over the joint action space by decomposing the joint value function Q into a sum of per-agent value functions:

$$Q((o_1, \dots, o_n), (a_1, \dots, a_n)) \approx \sum_{i=1}^n \tilde{Q}_i(o_i, a_i)$$

for n agents with observations o_1, \dots, o_n and actions a_1, \dots, a_n . Due to the monotonic relationship between \tilde{Q}_i and Q , maximising Q is now tractable by the independent maximisation of each

per-agent value function \tilde{Q}_i . QMIX extends this idea further with a more expressive decomposition function: rather than combine value functions via simple summation, they define a monotonic MLP (achieved through non-negative weights and an absolute value maximisation function) which processes the per-agent Q-values to produce the joint value. The weights for the MLP depend on the environment’s state via a HyperNetwork Ha et al. [2016]. This is a more expressive parameterisation, allowing the representation of a larger set of joint Q functions. However, the constraint of monotonicity is limiting; each agent acts based only on its local Q-function \tilde{Q}_i , and there is no way to represent the strength of joint actions. For example, imagine several agents controlling cars at a crossroads, each with actions stop or go. In this scenario, the joint value of any two agents simultaneously going (and crashing into one another) is very low - this cannot be represented by either VDN or QMIX.

This paper proposes an alternative *non-monotonic* decomposition of the joint value function, and the accompanying non-trivial maximisation algorithm.

2 Paired VDN

Both VDN and QMIX are restricted to representing joint Q -functions which decompose into a monotonic combination of per-agent Q -functions $\tilde{Q}_1, \dots, \tilde{Q}_n$, allowing the easy maximisation of Q as

$$\operatorname{argmax}_{a_1, \dots, a_n} Q((o_1, \dots, o_n), (a_1, \dots, a_n)) = \begin{pmatrix} \operatorname{argmax}_{a_1} \tilde{Q}_1(o_1, a_1) \\ \vdots \\ \operatorname{argmax}_{a_n} \tilde{Q}_n(o_n, a_n) \end{pmatrix}, \quad (1)$$

which also has the benefit of enabling decentralized execution: agent i just greedily maximises \tilde{Q}_i . Decentralization aside, eq. (1) enables the practical application of DQN to multi-agent settings with a large number of agents n , as it avoids the need to iterate directly over \mathcal{A}^n . In this project, we explore an alternative parameterisation of Q which does *not* satisfy eq. (1), allowing greater expressivity, but nevertheless allows *tractable* maximisation of Q in sub-exponential time in the number of agents. In particular, consider a decomposition of Q into n pair-wise functions $\tilde{Q}_{i,j}$, a method we name PairVDN, such that

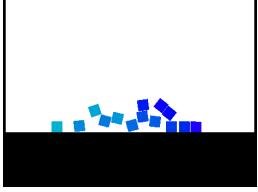
$$Q((o_1, \dots, o_n), (a_1, \dots, a_n)) \approx \sum_{i=1}^n \tilde{Q}_{i,j}((o_i, o_j), (a_i, a_j)) \quad (2)$$

where $j = i+1$ for $i < n$, wrapping around to $j = 1$ at $i = n$ (technically j is a function of i , but we leave this notation out for simplicity), and the indices $[1, 2, \dots, n]$ denote some arbitrary ordering of the agents. Intuitively, $\tilde{Q}_{i,j}$ gives some indication of the effect on future rewards of agents i and j jointly taking actions (a_i, a_j) . The terms form a cycle over the agents, with each action accounted for by two terms, rather than one in VDN. PairVDN is strictly more expressive than VDN, as the network may learn $\tilde{Q}_{i,j}((o_i, o_j), (a_i, a_j)) = \tilde{Q}_i(o_i, a_i)$, but is neither more nor less expressive than QMIX.

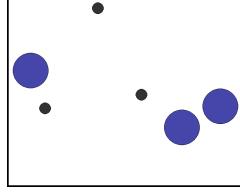
Intuitively, our formulation allows agent actions to take into account those of other agents at the same timestep. For example, imagine an environment in which all agents must work together to break a wall, and all must simultaneously hit the wall for the best result. PairVDN is able to effectively represent this value function by assigning a positive weight to $\tilde{Q}_{i,j}((o_i, o_j), (a_i, a_j))$ only when $a_i = a_j$. This function is clearly maximised for $a_1 = a_2 = \dots = a_n$, causing the agents to work together and act in unison as is required. This example value function does not satisfy (2) for any set of per-agent Q-functions.

Practically, this may be implemented in a very similar manner to VDN, but with each neural network $\tilde{Q}_{i,j}$ receiving two observations as input and outputting a vector of length $|\mathcal{A}|^2$, which is then reshaped into an $|\mathcal{A}| \times |\mathcal{A}|$ grid. The n networks $\tilde{Q}_{1,2}, \dots, \tilde{Q}_{n,1}$ may share parameters as with VDN and QMIX.

As each action a_i occurs in exactly two terms in (2), maximising Q is no longer trivial, and may not be achieved in a decentralized manner. We achieve efficient maximisation through a dynamic programming algorithm, with time complexity $\mathcal{O}(n|\mathcal{A}|^3)$, scaling *linearly* with the number of agents, although poorly with $|\mathcal{A}|$. This algorithm is asymptotically superior to naive iteration over the joint



(a) Box Jump



(b) Simple Spread



(c) CookingZoo

Figure 1: The three environments we evaluate on. Box Jump is a custom environment we design for this project.

action space, which requires $\mathcal{O}(|\mathcal{A}|^n)$ time, and quickly becomes infeasible as n grows. Full algorithm details may be found in Appendix A.

3 Experiments

3.1 Environments

Limited to fully co-operative environments with discrete action spaces, with a preference for simple environments with support for many agents (and global state - required by QMIX as input to the hypernetworks), we select the following environments:

- Simple Spread: a simple inbuilt environment in which agents must get as close as possible to target locations without touching one another.
- Cooking Zoo Rother et al. [2023]: a third-party environment inspired by the videogame Overcooked and adapted to PettingZoo. We modify this slightly to add support for global state and an easier level.
- Box Jump: a custom environment we designed during the project. It is a relatively simple 2D physics-based environment, in which each agent is a box, and agents are rewarded based on the maximum height of any agent, incentivising them to work collaboratively to build a tower. It is designed with a large number of agents in mind, and runs well with 16 or more. We emphasise that PairVDN has no inherent advantage on Box Jump; it was not designed to favour our approach, but to allow scaling to many agents. Full details in Appendix B.

The environments are shown in Figure 1. In each case, agents receive only a shared global reward signal. We stress that this makes each environment highly challenging, especially with a large number of agents, due to the credit assignment problem.

3.2 Setup

Each joint value function $\tilde{Q}_{i,j}((o_i, o_j), (a_i, a_j))$ is parameterised as a multi-layer perceptron with ReLU activation, input dimension $2|\mathcal{S}|$, and output dimension $|\mathcal{A}|^2$ giving the estimated Q-values for each pair of actions $(a_1, a_2) \in \mathcal{A}^2$. The parameters of the MLP are shared across all agents to accelerate training, with one-hot agent identifiers appended to the observations. Our codebase is implemented from scratch using the jax, equinox and optax libraries.

As suggested in DQN Mnih et al. [2015] we employ a modified objective based on a target network with parameters θ^- . The target network is updated via the exponential moving average,

$$\theta_{t+1}^- = c\theta_t^- + (1 - c)\theta$$

following the t_{th} update of the parameters θ . We employ an ϵ -greedy policy to encourage exploration, with the value of ϵ interpolated during training from $\epsilon = 1$ to $\epsilon = 0.05$. Full hyperparameter details may be found in Appendix C.

We compare PairVDN to VDN, QMIX, and independent Q-learning (IQ-L). We find it beneficial to initialise QMIX in a HyperNetwork-specific manner Chang et al. [2023].

3.3 Results

Figure 2 shows rewards during training for three Box Jump tasks, and the cooking and simple spread tasks. All models perform similarly on the 8 agent Box Jump task, but when the number increases to

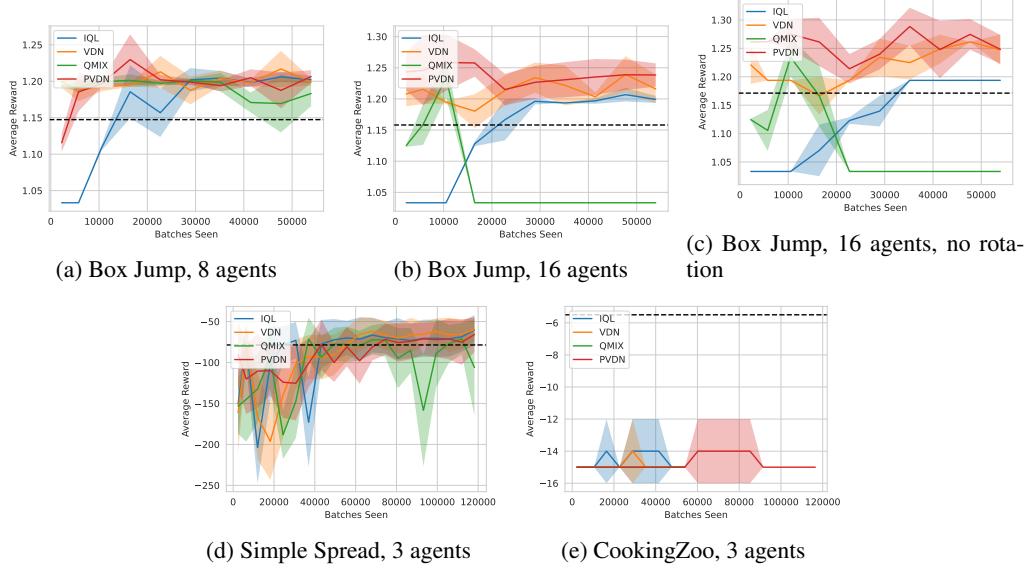


Figure 2: Total episode rewards during training for the various trained models, with standard deviation across five episodes shown shaded. Black dashed line gives performance of a random baseline (averaged over 30 runs). PVDN denotes PairVDN.

Table 1: Average total reward over 20 episodes on Box Jump with 16 agents after training. VDN performs similarly to PairVDN after 400 steps, but after 1000 there is a clear difference. Best mean result in bold, although the standard deviation should be taken into account.

	Rotation		No Rotation	
	$t_{\max} = 400$	$t_{\max} = 1000$	$t_{\max} = 400$	$t_{\max} = 1000$
Random	1.170 ± 0.025	1.197 ± 0.013	1.178 ± 0.023	1.225 ± 0.032
IQL	1.211 ± 0.019	1.212 ± 0.016	1.194 ± 0.000	1.228 ± 0.045
QMIX	1.033 ± 0.000	1.033 ± 0.000	1.033 ± 0.000	1.033 ± 0.000
VDN	1.224 ± 0.028	1.235 ± 0.028	1.244 ± 0.029	1.258 ± 0.026
PairVDN	1.239 ± 0.031	1.271 ± 0.019	1.259 ± 0.026	1.294 ± 0.032

16, QMIX becomes unstable, and IQL becomes ineffective, with the difference further exaggerated in the variant without box rotation.

Table 1 gives detailed results for each model on the Box Jump task with 16 agents, with an additional column $t_{\max} = 1000$ measuring the performance on longer episodes. Note that increasing the time limit can only feasibly increase a model’s score, as it has more time to build a larger tower. While Figure 2 seems to show PairVDN achieving similar performance to VDN, the $t_{\max} = 1000$ experiments reveal a significant difference: due to its non-monotonic aggregation, PairVDN leads to far more *coordinated* agent behaviour, causing agents to remain clumped together for a long period of time, while they slowly disperse under VDN (Figure 3). This is evident in the magnitude of the improvement under PairVDN when increasing t_{\max} from 400 to 1000, which is significantly larger than for any of the baselines. We consider this improved coordination to be a significant finding of our work, highlighting the weakness of monotonic value decomposition.

All models perform very poorly on the cooking environment. This is a challenging, complex environment, with verbose observations (dimensionality 119) and sparse rewards - further worsened by the loss of specific per-agent rewards. It is evidently too complex for our simple DQN architecture, and likely requires the use of more complex agent architectures, such as those containing recurrent units. The models additionally perform poorly on Simple Spread, achieving scores little better random, which is also likely due to the simple network architecture and small MLP size.

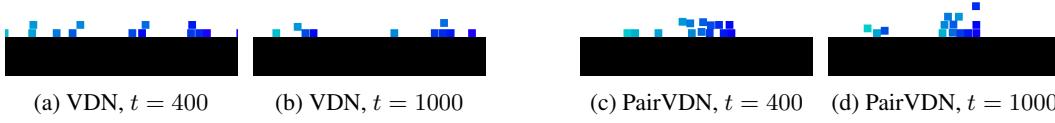


Figure 3: Behaviour of VDN and PairVDN on Box Jump with 16 agents, no rotation, and seed zero in both cases. Notice how PairVDN’s agents are grouped together more closely than VDN’s at time $t = 400$, and more so at time $t = 1000$.

4 Conclusion

This paper proposes PairVDN, a novel decomposition of joint multi-agent value functions, which is notably able to represent *non-monotonic* value functions. We further provide the non-trivial dynamic programming algorithm which is necessary to efficiently optimise PairVDN, and include a differentiable implementation in our public codebase. The terms effectively form a loop over the agents, with each agent’s action accounted for by two terms, leading to improved cooperative behaviour after global maximisation. Finally, we demonstrate the efficacy of our method on a complex environment with a large number of agents, despite the simple DQN architecture, and achieve superior performance to VDN, QMIX and IQL.

References

- Oscar Chang, Lampros Flokas, and Hod Lipson. Principled weight initialization for hypernetworks, 2023.
- David Ha, Andrew Dai, and Quoc V. Le. Hypernetworks, 2016.
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 00280836. URL <http://dx.doi.org/10.1038/nature14236>.
- Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Monotonic value function factorisation for deep multi-agent reinforcement learning. *J. Mach. Learn. Res.*, 21(1), January 2020. ISSN 1532-4435.
- David Rother, Thomas Weisswange, and Jan Peters. Disentangling interaction using maximum-entropy reinforcement learning in multi-agent systems. In *European Conference on Artificial Intelligence*, 2023.
- Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. Value-decomposition networks for cooperative multi-agent learning based on team reward. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS ’18, page 2085–2087, Richland, SC, 2018. International Foundation for Autonomous Agents and Multiagent Systems.

A PairVDN Dynamic Programming Algorithm

We wish to maximise F where

$$F(a_1, \dots, a_n) = \sum_{i=1}^n \tilde{Q}_{i,j}((o_i, o_j), (a_i, a_j)) \quad j = \begin{cases} i+1 & i < n \\ 1 & i = n \end{cases}$$

The cyclic dependency makes this slightly more difficult, and increases the runtime of the algorithm from $\mathcal{O}(n|\mathcal{A}|^2)$ to $\mathcal{O}(n|\mathcal{A}|^3)$. Informally, we solve the non-cyclic version (without $\tilde{Q}_{n,1}$) for each possible starting action a_1 , then use these to find the global optimum.

For $2 \leq k \leq n$, define

$$G_k(a_1, a_k) = \max_{a_2, \dots, a_{k-1}} \sum_{i=1}^{k-1} \tilde{Q}_{i,j}((o_i, o_{i+1}), (a_i, a_{i+1}))$$

That is, the max sum achievable over the first k terms with the given values of a_1 and a_k . Note that the upper bound on this sum is $k - 1$, so G never includes the ‘cyclic’ term $\tilde{Q}_{n,1}$. It follows from this definition that

$$G_{k+1}(a_1, a_{k+1}) = \max_{a_k} \left(G_k(a_1, a_k) + \tilde{Q}_{k,k+1}((o_k, o_{k+1}), (a_k, a_{k+1})) \right),$$

and let $P_{k+1}(a_1, a_{k+1})$ be the corresponding argmax. When $k = 2$ we have $G_2(a_1, a_2) = \tilde{Q}_{1,2}((o_1, o_2), (a_1, a_2))$. Using these two equations, G may be implemented as an array with $n|\mathcal{A}|^2$ entries, and calculated completely in time $\mathcal{O}(n|\mathcal{A}|^3)$, the bottleneck being the iteration over a_k in the equation above.

Finally, we must ‘close the loop’ and relate G back to F :

$$\max_{a_1, \dots, a_n} F(a_1, \dots, a_n) = \max_{a_1, a_n} \left(G_n(a_1, a_n) + \tilde{Q}_{n,1}(a_n, a_1) \right)$$

The argmax here gives the optimal values for a_n and a_1 , from which we may backtrack (using P) to construct the complete optimal sequence of actions.

A.1 Future Extensions to PairVDN

Inspired by QMIX, it may be beneficial to incorporate state-dependent weights w_i for each $\tilde{Q}_{i,j}$ to improve expressivity further - in fact, these weights may be negative here, as PairVDN does not require monotonicity, unlike QMIX. The algorithm above could be easily adapted for this.

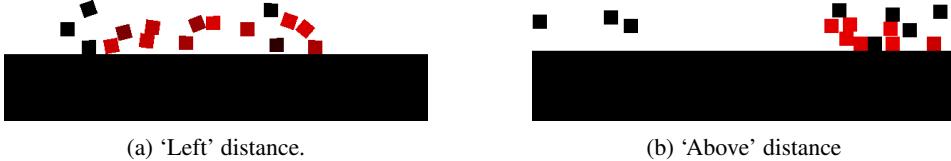
Additionally, PairVDN is limited by its arbitrary ordering over agents, with terms covering a fixed cycle $j = i + 1$. As the relevance of each agent pair is likely to change during an episode, we could instead allow the choice of j for each i to change at each step (for example, by choosing the agent currently closest to i). This changes the graph from a single directed cycle into a more complex directed structure, with n nodes and n edges. Such a graph is guaranteed to consist of some number of closed loops, containing m nodes in total, with the remaining $n - m$ forming directed trees with their root on one of these cycles. The algorithm can be adapted for these trees - computation is faster on a tree due to the lack of cyclic dependencies, and is possible in $\mathcal{O}(n|\mathcal{A}|^2)$ via a similar algorithm to that above. For each tree, which is guaranteed to have a root on a cycle, the algorithm computes the best score achievable across the whole tree for each action taken at the root. Following this computation, we may run a slightly modified version of the algorithm above on each cycle in the graph individually to again retrieve the global maximum in time $\mathcal{O}(m|\mathcal{A}|^3 + (n - m)|\mathcal{A}|^2) = \mathcal{O}(n|\mathcal{A}|^3)$.

B BoxJump Environment Details

The action space has size $|\mathcal{A}| = 4$, where 0 is a no-op, 1 and 2 apply forces left/right, and 3 jumps (if the box is able to jump, otherwise does nothing). Agents start each episode exactly at ground level, spaced evenly across a horizontal line, with a slight random variation in position to prevent .

The observation o_i for each agent consists of:

- Position: (x, y) , each ranging from 0 to 1.



(a) ‘Left’ distance.

(b) ‘Above’ distance

Figure 4: Example values of the left (a) and above (b) distance observations, where red indicates a value of zero, black a value of one, with a gradient between. Notice how the ‘above’ distance observation clearly indicates to the agent whether another agent is stacked on top of it (when the value is zero).

- Velocity: (v_x, v_y) , given in position units per second.
- Angle: $\theta \in (-0.5, 0.5]$, giving the number of *quarter rotations*. As rotating a box by 90 degrees creates an identical state, there is no need to distinguish between these states. $\theta = 0$ indicates that the box’s sides are parallel to the floor, while 0.5 indicates 45 degrees of rotation.
- Left/right distances: the distance that a ray fired to the left/right travels before hitting another box. More precisely, this is the horizontal distance to the closest box with which this box overlaps vertically.
- Up/down distances: as with left/right. Down distance effectively tells the agent whether it is mid jump or stationed on the floor, while up distance tells the agent whether another agent is standing on top of it.
- Whether the box can currently jump.
- Highest y value observed this episode (by any agent) up until now, y_t^{best} .
- The remaining time t . This is necessary for the environment to satisfy the Markov condition.

Figure 4 demonstrates some example observation values visually.

We use a sparse reward signal, rewarding all agents evenly only when a new global maximum height is achieved. Formally, $r_t = y_t^{\text{best}} - y_{t-1}^{\text{best}}$, where $y_t^{\text{best}} = \max_{i,t' \leq t} y_{i,t'}$, incentivising the building of tall towers. Note that this scheme causes the total reward of an episode to be the best y value achieved over the whole episode, $\sum r_t = y_T^{\text{best}}$ where T is the total episode length.

B.1 Limitations

- Boxes are allowed to jump iff their vertical velocity has not exceeded a small threshold in the last 15 frames. This leads to a slight delay in their jumping, and could theoretically be exploitable.
- The observation calculation code has quadratic runtime with the number of agents due to the left/right/up/down distances - this could be improved using a more complex sweep-based algorithm.
- Agents can walk off the sides of the map, then fall infinitely. There is a large reward penalty to handle this.

C Hyperparameters

Hyperparameter	Value
Epochs	100
Learning rate	1e-4
Batch size	32
Gamma γ	0.99
Target network c	0.99
ϵ -greedy value	$1 \rightarrow 0.05$
MLP layer sizes	$2 \mathcal{S} , 128, 128, \mathcal{A} ^2$
Exploration per epoch	400
Optimiser	SGD
Experience buffer size	20,000

Table 2: Hyperparameter values.

D Additional Figures

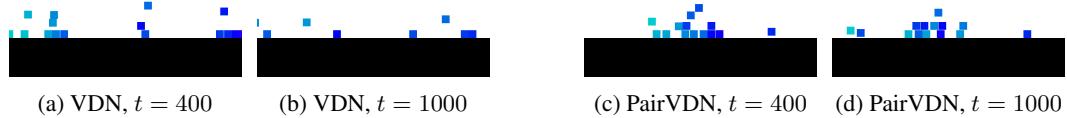


Figure 5: Figure 3 but for seed 1 rather than seed 0, demonstrating that the figure was not cherry-picked but a frequent phenomenon.