The Reimplementation and Application of Direct Future Prediction to Simple Environments and Online Learning

Michael Aboody, Jongmin Jerome Baek, Yu-chieh Lee*
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

December 2017

Contents

1 Problem Definition and Motivation **2**

2 Related Work and Comparisons **2**

3 Approach **3**
3.1 Overview 3
3.2 Experiences and Temporal Offsets 3
3.3 Specifics of the Agent 3
3.4 Network Specifics 4
3.5 Simulator Specifics 5
3.6 Offline Potential (Extended Analysis) 5

4 Implementation Details **6**
4.1 Bounded Cache for Experience Creation/Experience Memory 6
4.2 BasicNetwork Masking & Details 6
4.3 Preprocessing Observations for the Network 6
4.4 Using a "Dummy" Network Module for Testing 7
4.5 AWS 7
4.6 Offline Serialization/Deserialization 7

5 Results **8**
5.1 Method of Evaluation 8
5.2 Replication of Original Results 8
5.3 Extended Analysis: Flappy Bird 8
5.4 Extended Analysis: Offline Learning 9

6 Code and Media **10**

*{mikeaboody,jbaek080,14leeyuchieh}@berkeley.edu

1 Problem Definition and Motivation

Koltun et al. present a novel machine learning architecture in "Learning to Act By Predicting the Future" [DK16]. In this report, we test the flexibility and general applicability of Koltun's architecture. To this end, we:

(1) Reimplement the code in Keras [Cho+15] and test it on VizDoom [Kem+16] to reproduce results;

(2) Apply the model to a substantially simpler environment, the game of Flappy Bird;

(3) Test the offline capabilities of Koltun's model, training an offline agent with a set of observations and actions from an expert agent.

2 Related Work and Comparisons

Direct Future Prediction (DFP), the model we implement in this report, has a number of related models. Here we explicate a few relevant papers which, like DFP, also tackle the ViZDoom Challenge.

In "Playing Doom with SLAM-Augmented Deep Reinforcement Learning" [Bha+16], DQN-SLAM (from here just DQN) uses object categorization and localization to tackle the ViZDoom challenge. We compare and contrast DQN with DFP. First of all, DQN is a model inspired by recent findings in human cognition, while the same cannot be said of DFP. DQN replicates some functionalities of human cognition, such as object detection, 3D reconstruction, etc., in separate models. DFP, on the other hand, is designed almost exclusively to predict the future, and therefore does most pre-processing, identification, etc., implicitly in one convolutional neural net. This may be why DFP achieved a better result than the DQN model. Second of all, DFP has a much higher action space: it has 256 possible actions (in the Battle 2 Experiment) as opposed to the 36 possible actions for the DQN model. Third of all, the DFP model was trained in multiple simulators simultaneously, whereas the DQN model was trained in just a single simulator. The differences in the training environment and action space, and resulting differences in flexibility, as well as differences in the complexity of the environments, may have further caused the difference in results.

We also examined the DQN and LSTM-A3C [Mni+16] approach in the ViZDoom environment in "Deep Reinforcement Learning From Raw Pixels in Doom." [Haf16] We notice that, in this paper and [LC16], the AI was not trained for as long as detailed in the actual experiment, and therefore it may not reflect the effectiveness of DQN and LSTM-A3C fully. In both DQN and LSTM-A3C, it seems that the agent does not remember the opponents after seeing them, and it often correctly turns toward enemies as well, but fails to aim correctly. The DQN and LSTM-A3C models do not reward health packs, and therefore both models often navigate along walls and go into rooms for weapons, while ignoring rooms with health packs. Notice that the DFP model solves these problems by considering all sets of actions and experiences that it is being trained upon, and given enough training, is in principle able to fit well to its environment without an artificially set reward. DFP does not necessarily remember its opponents, but it may, by merit of it learning from all its training data to predict the future, and so it may remember cases where opponents sneak up on it. Therefore, we may say that DFP is an improvement over DQN.

By the time the project was finished, there was no existing code source of Koltun's model outside of the original implementation in TensorFlow. On November 17, 2017, we

noticed that a new, unfinished re-implementation in Keras [Yu17] was published. This paper, however, does not detail everything that the original paper did, nor is it clear if it concretely replicated results of the original paper. We mention this re-implementation purely to avoid confusion.

3 Approach

3.1 Overview

A DFP agent takes in an **observation** (a high dimensional **sensory input** plus a low dimensional **input measurement**), learns how that measurement changes in certain future timesteps (called **temporal offsets**) for each possible action, and determines the action that yields the best changes to that measurement based on some given **goal**. In essence, this is how a DFP agent acts. Making the right decision, however, is all about *observing* how these measurements change based off of the agent's observations and actions. This process (and thus our code) can effectively be abstracted into 3 interdependent modules: the agent, the network, and the simulator. In this process, the agent is the middleman. The simulator interacts with the agent both by producing images/measurements of the current step's environment for the agent to use in observing/acting and by receiving an optimal action from the agent for a particular observation. When the agent observes, the agent keeps recent *experiences* and periodically feeds those experiences to the network to update the network's ability to judge future measurements based off of an observation, action, and goal. Upon finding the optimal action, the agent queries the network for the resulting change in measurement based off some given observation/goal for each action, and the agent determines which is the best for the goal.

3.2 Experiences and Temporal Offsets

An experience can be thought of as a training point used to learn how to predict future measurements. The X_i vector, or one column of the training data matrix, is an observation (represented by the `Observation` class) for a timestep, the action taken at that timestep (a one hot encoded vector), and the current goal (as a vector) used to train at the timestep. The Y_i vector, or one column of the labels matrix, is how that measurement changes with time. This notion of "future measurements" is encapsulated in what is known as temporal offsets. Temporal offsets represent future offsets in time from the current timestep where future measurement will be compared to the current measurement. This means that for a perceived measurement at time t called m_t, the difference between measurements at time t and at time $t+i$, which are $m_{t+i} - m_t$, will be stored for each temporal offset in a numpy array. For example, in doom, we use health as a measurement, and we have temporal offsets of 1, 2, 4, 8, 16, and 32 (as specified in the paper). This means we will track how the measurement changes 1, 2, 4, 8, 16, and 32 steps ahead. Experiences are the basic abstraction for communication between the agent and the network. These quantities are stored in instances of the `Experience` class.

3.3 Specifics of the Agent

The agent, represented in our code by the "Agent" class, has two primary methods: `observe(...)` and `act(...)`.

The `observe(...)` method is called during training with an input observation (represented by the Observation class)/action pair at the current time step and returns nothing. The agent first uses an `ExperienceCreator` to build an experience. The `ExperienceCreator` keeps track of the most recent observation/actions observed in this episode, with the amount corresponding to the largest temporal offset (in our case that was 32). For the least recently kept observation/action pair, the changes in its measurements at each temporal offset is calculated (only if a measurement at a temporal offset in the future has been observed), resulting in a label for that time step from which an experience can be created from. Providing the `ExperienceCreator` could create an experience, it would be added to whats known as the experience memory (which is implemented through the `BoundedCache` class), or the last M experiences created (which was 12500 for doom). Every k experiences that are created (32 in the paper), $N = 64$ experiences from the experience memory are sampled to be used in updating the weights of the network in a minibatch. If the new experience created results in this occurence, the `sample(...)` method is called on the `BoundedCache` instance for N experience, and these are fed to the network via the `Network` class's `update_weights()`.

The `act()` method is called during training/testing with an input observation plus whether or not its training and returns the optimal action for that observation. Particularly during training, the actions produced by the agent follow a random schedule (similar to deep reinforcement learning) where with some probability ϵ a completely random action is taken. As training goes on, ϵ is decreased until it is 0. If the agent is training, a random action will returned with probability ϵ. Otherwise, it will query the network using the `Network` class's `predict()` method, which will return the predicted change in future measurements from a given observation and goal for each possible action. Then, using the inputted testing goal vector, this vector is dotted with each action's predicted changed in future measurements, and the action with the highest result will be chosen as the optimal action.

3.4 Network Specifics

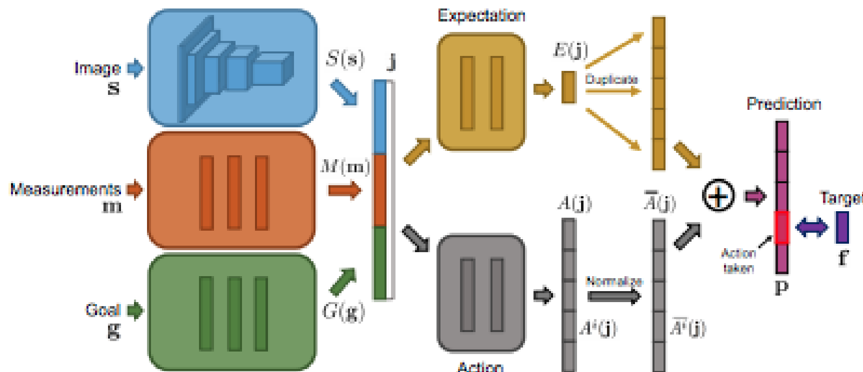

Figure 1: DFP Network Architecture by Koltun et al.

The paper essentially models Deep-RL, [LC16] specifically the DQN model, as a supervised learning problem. The preprocessing and agent components are essentially the same. The neural network takes the image, measurement, and goal as input into the network. The image goes through a CNN, the measurement through a FCC-NN, the goal through a FCC-NN. The vector outputs of these three neural network then goes into an expectations NN and an action NN. The interpretation for the expectation network is the average future measurements over the future. The action network outputs the fine differences in those measurements among different actions. The results of these two network are then finally added together to create the prediction vector. The prediction vector is interpreted as the states over several temporal offsets into the future for each possible action the agent can take concatenated. Our loss function is the MSE only for the specific action we've taken during training, not for the entire prediction vector.

3.5 Simulator Specifics

Regardless of the application, the simulator needs to be able to take in an action and provide the resulting sensory input/measurement at a particularly timestep. Particularly, the Doom simulator was implemented using the VizDoom library, which allows one to simulate episodes of doom and provides basic data. The Doom simulator we provide is heavily based off of the paper's implementation of the simulator provided in the paper. Effectively, its step() method takes in an action vector and collects an 84×84 grayscale image each step and the player health at each time step (as well as if the episode terminated). In addition, it implements "frame skipping", which means the same action is applied every 4 frames and data is only collected every 4 frames.

One thing done in the paper which was somewhat necessary to do in this reimplementation was to train on multiple simulators at once. In particular, the agent would be making observations from each of the simulators during training and act() was called for each of the simulator's requests. This would not speed the program up (as this was done sequentially), but its purpose was instead to diversify the set of experiences in the experience memory. The results of the experiments greatly improved when this implementation was added.

3.6 Offline Potential (Extended Analysis)

Throughout the training procedure, the DFP agent is trained online by sampling its recent experiences for minibatch updates. We conjectured that a DFP agent may be capable of offline learning via taking a large set of experiences from some agent, which need not necessarily be a DFP agent, and by letting the offline-learning DFP agent train on that set of experiences. Theoretically, if the model learns to predict correctly by offline training, the agent could make good decisions. For our extended analysis, we look at training an agent offline from another agent's experiences. The purpose of this analysis was to determine the model's reliance on its own experiences/actions to predict measurements accurately.

We worked with 3 different agents for training offline:

1. A completely random agent;

2. A trained, expert DFP agent;

3. An "mixed" agent, which chooses a random action half the time and an expert DFP agent's suggested action half the time.

4 Implementation Details

4.1 Bounded Cache for Experience Creation/Experience Memory

Experience creation and experience memory both require a datastructure that acts like a cache for just the last k items. When this cache is full, it removes the least recently added item. In addition, this cache must support finding the i-th most recently added item, random sampling (with replacement) of n items, and reporting when its at capacity (has k items).

This yielded our `BoundedCache` datastructure. It stores an initially empty list that grows as more items are added until k items are held. Then, it tracks the index of the next item to "evict", and replaces its entry when a new item is updated. This index can also be used to determine easily which items were least recently added. The benefit of this approach is not having to allocate a new array upon addition of each new item (which might have improved performance, but it's hard to say).

The datastructure has a few main methods:

1. add(item): adds an item and returns the evicted item if an item has been evicted
2. sample(n): samples at random n random entries from the cache
3. index_from_back(i): returns the i-th least recently added item
4. at_capacity(): determines if k items have been added

The `ExperienceCreator` must maintain that last 32 observation/action pairs it has seen. To do so, it uses a `BoundedCache` of capacity 32 to maintain these pairs, and uses index_from_back(i) to get the appropriate items to build the label and thus the experience. Furthermore, the experience memory must maintain the last 12500 items it finds and occasionally sample with replacement from these items. This could easily be implemented through a `BoundedCache`, and experiences could be easily sampled using the sample(n) method.

4.2 BasicNetwork Masking & Details

We added a hack to Keras in order to make the model's loss function work. The model's loss function is MSE specifically for 1) the expected vector of the states over the specified temporal offsets for the action taken and 2) the predicted vector of the states over the specified temporal offsets for the action. So we had a mask vector added into the network that is 1 for any component that relates to the action taken and zeroed out for everything else in the predicted vector.
Using truncated normal weights and exponential decay over Keras' default settings were helpful to results in the VizDoom health pack experiment. Specifically, we set our network layer's weights to truncated normal weights and we had an exponential decay function for our learning rate based on the steps of training we did.

4.3 Preprocessing Observations for the Network

It sometimes is necessary to preprocess what is fed into the network to allow it to train more easily. For example, in Doom, the sensory input is an 84×84 grayscale image ranging from 0-255 and the measurement is a scalar ranging from 0-100. Feeding these directly into

the network doesn't exactly work with the hyperparameters mentioned in the paper since the units of the network become easily oversaturated. To compensate, it was necessary to preprocess the image and health attributes before feeding it into the network by normalizing them to have ranges between -0.5 and 0.5. For generalization, we provide configurable preprocess methods for both the sensory input and the measurement in the Doom config to allow for this preprocessing. Note that we ONLY want to preprocess these right before they're fed into the network, or the experience creation procedure will yield labels with too small entries.

4.4 Using a "Dummy" Network Module for Testing

When running the entire process end to end, it's difficult to determine whether there a bug in the algorithm exists. Everything can be implemented correctly, but if the network's hyperparameters are even somewhat off, it will not function. Particularly, we wanted to isolate whether it was the network that was the issue or if it was something else. Since we abstracted well, we could simply create a "Dummy" Network class called `BlankNetwork`, which has the same methods as the `BasicNetwork` class, but instead simply prints out debugging information about what has been passed in. When we used the `BlankNetwork` instead of the `BasicNetwork`, we could more easily see that the algorithm was working properly since the debugging output looked the way it was expected to. This allowed us to isolate the issues to `BasicNetwork`, and thus we focused on our attention on fixing that class (which ended up needing a lot of fixing).

4.5 AWS

Running this complicated network structure on our local computers was simply infeasible given how slow both predicting and minibatch updates would work in Keras on a CPU. On our computers, we calculated it would take 10 days simply to run 1 million iterations. To address this slowness, we rented an AWS instance called `p1.xlarge`, which has a Tesla K80 GPU, 60GB of RAM, and 100GB of storage. This sped up computation significantly, allowing for training on 5 million steps in 36 hours.

4.6 Offline Serialization/Deserialization

To implement our offline extended analysis, we had to find a way to store our entire dataset of experiences from 5 million steps. This yielded a huge problem: just for one agent's set of experiences, 5 million steps corresponded to 30GB of data. This would definitely not work in RAM. To address this issue, we created an "experience serializer" and an "experience deserializer". Effectively, the serializer would buffer experiences to files on disk in chunks (specified by the user) into a folder. Then, the "experience deserializer" would yield a generator that would efficiently load each of these small chunks into memory (one at a time) and yield each experience as a python generator. This meant that while we had 30GB of data for each set of experiences we trained on, only roughly about 80MB of data were in memory at any time.

5 Results

5.1 Method of Evaluation

Evaluation of the model consisted of training on five million steps, or frames. We trained on the “Basic” Doom map, which consists of an agent running around in a room, periodically receiving damage, and collecting health packs in order not to die. We performed a testing procedure periodically, once every 100,000 training iterations. Each testing procedure included a hundred test episodes where the terminal health of each episode was tracked. After collecting 100 terminal health statistics each trial, the statistics were averaged and plotted for each 100,000 steps, and compared to the results found in the paper.

5.2 Replication of Original Results

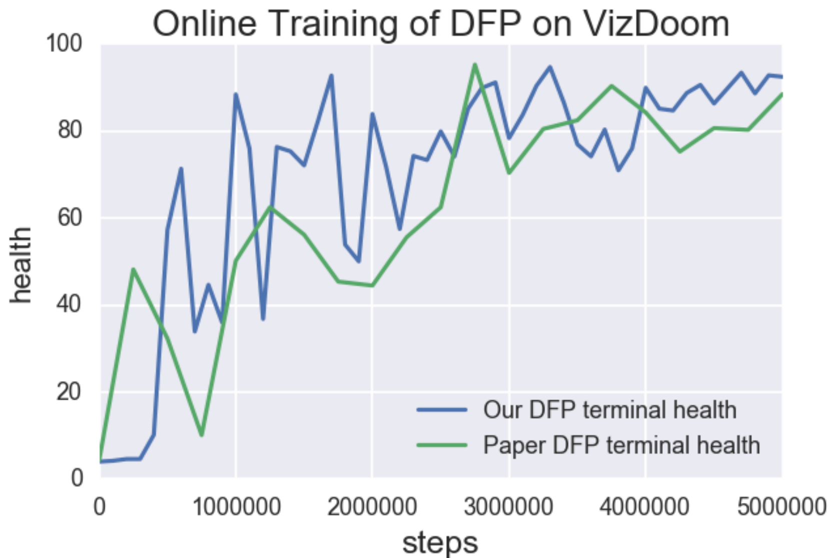

Figure 2: Online Training of DFP on VizDoom

The diagram illustrating this comparison is shown above. Our DFP agent’s performance roughly mirrors Koltun’s DFP agent’s performance for the first five million steps of training. As can be seen in the attached video[1], the agent was trained to avoid walls and locate health packs. Therefore, we successfully replicated Koltun et al.’s results to a reasonable degree of accuracy.

5.3 Extended Analysis: Flappy Bird

Flappy Bird has been demonstrated solved by a DQN network [Lau16]. However, DFP didn’t generalize to Flappy Bird despite repeated hyperparameter tuning. We attempted the following: (1) varied the goal vector to weight immediate futures more heavily, (2)

[1] See section 6, “Code and Media”, for this video.

modified the measurement to include bird velocity as well as the score, and (3) modified the number of frames in a single perception from 4 to 1. Despite our attempts, the model did not generalize. We suspect at least two reasons. First, the measurement signal, as indicated by the number of pipes the bird jumped through, was highly discrete. Second, the agent had little time to train before crashing and would quickly converge to a local minimum.

5.4 Extended Analysis: Offline Learning

As specified in our approach, we trained an offline agent for each set of experiences yielded by 3 different agents:

1. A completely random agent,
2. Our trained DFP agent
3. An agent that chooses a random action half the time and our DFP agent's suggested action the other half.

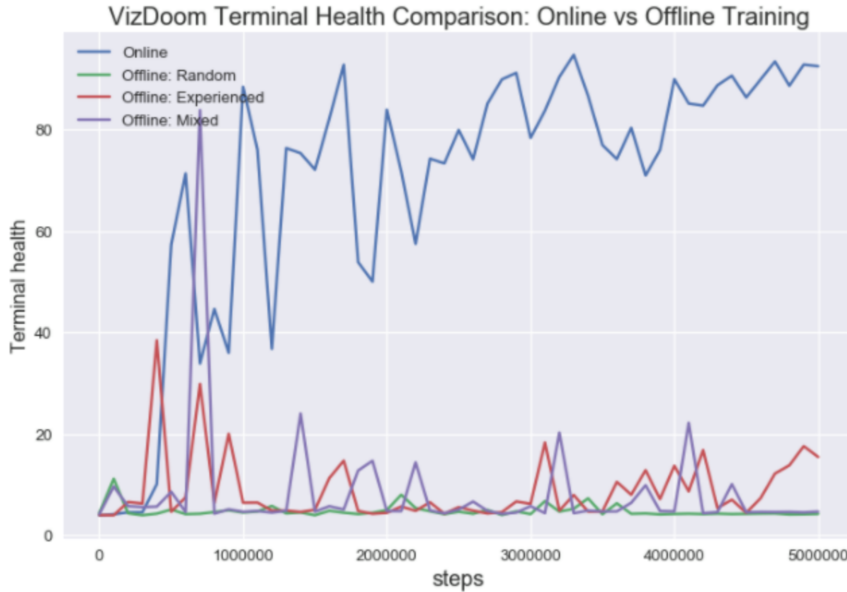

Figure 3: VizDoom Terminal Health, Online vs. Offline

Each of these agents were run for five million steps and experiences were collected for each. Then, offline learning was performed on each set of experiences with the same hyperparameters as the online method and testing trials were periodically run in the same manner as in the online training, but we also tracked mean health through each episode. The mean health and terminal health results for each set of experiences were plotted along with the results from online learning.

However, our offline learning approach did not converge to a reasonable solution. As the diagrams above illustrate, none of the three offline learning approaches yielded a model

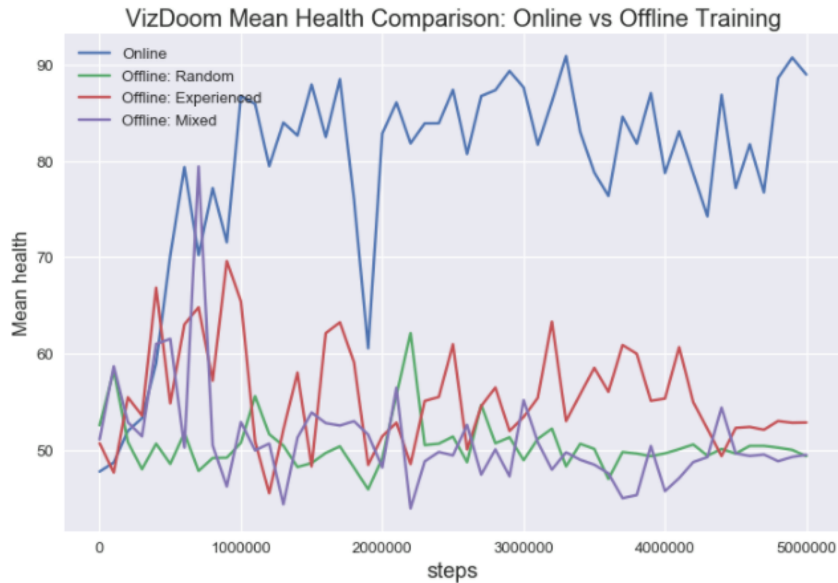

Figure 4: VizDoom Mean Health, Online vs. Offline

comparable to our online learning model. We suspect several reasons for this limitation. It is well-known [RGB10] that naïve imitation learning rarely yields an agent comparable in performance to the original agent, because small discrepancies in observation and action quickly accumulate and cause the imitation learning agent's policy to quickly (that is, exponentially) diverge from the expert policy.

6 Code and Media

In addition to our Github repository, we are providing a video of our online-trained DFP agent in action. Note how well our agent avoids walls and collects health packs.

Code:
`https://github.com/mikeaboody/DFP-Code-Reimplementation`

Video:
`https://youtu.be/x0QxFP2Ahiw`

References

[RGB10] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. "No-Regret Reductions for Imitation Learning and Structured Prediction". In: *CoRR* abs/1011.0686 (2010). arXiv: 1011.0686. URL: http://arxiv.org/abs/1011.0686.

[Cho+15] François Chollet et al. *Keras*. https://github.com/fchollet/keras. 2015.

[Bha+16] Shehroze Bhatti et al. "Playing Doom with SLAM-Augmented Deep Reinforcement Learning". In: *CoRR* abs/1612.00380 (2016).

[DK16] Alexey Dosovitskiy and Vladlen Koltun. "Learning to Act by Predicting the Future". In: *CoRR* abs/1611.01779 (2016). arXiv: 1611.01779. URL: http://arxiv.org/abs/1611.01779.

[Haf16] Danijar Hafner. "Deep Reinforcement Learning From Raw Pixels in Doom". In: *CoRR* abs/1610.02164 (2016).

[Kem+16] Michał Kempka et al. "ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning". In: *IEEE Conference on Computational Intelligence and Games*. The best paper award. Santorini, Greece: IEEE, Sept. 2016, pp. 341–348. URL: http://arxiv.org/abs/1605.02097.

[LC16] Guillaume Lample and Devendra Singh Chaplot. "Playing FPS Games with Deep Reinforcement Learning". In: *CoRR* abs/1609.05521 (2016).

[Lau16] Yanpan Lau. *Flappy Bird Keras*. https://yanpanlau.github.io/2016/07/10/FlappyBird-Keras.html. 2016.

[Mni+16] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[Yu17] Felix Yu. "Direct Future Prediction - Supervised Learning for Reinforcement Learning". In: (2017). URL: https://flyyufelix.github.io/2017/11/17/direct-future-prediction.html.