AI Programming (IT-3105) Spring 2025
Main Project:
A MuZero Knockoff

Due Date: NOON on Friday, May 2, 2025 (Last day to upload group videos to Blackboard)

Purposes:

1. Gain further familiarity with Python's JAX package for automatic differentiation (or a similar package such as PyTorch).
2. Get an in-depth, hands-on understanding of one of Google DeepMind's most successful game-playing systems.
3. Learn to produce an educational, technical video.

1 Introduction

In 2019, Google DeepMind followed up their earlier ground-breaking systems (AlphaGo, AlphaGo Zero, and Alpha Zero) with one that required even less *human expertise*: MuZero. Whereas AlphaGo used the results of many professional-level Go games to train early neural networks (that provided the starting point for self-play), AlphaGo Zero skipped that preliminary step and simply started self-play with completely novice-level agents. This eventually yielded a Go player that beat AlphaGo 100 games to 0. They then generalized AlphaGo Zero to play a few other games (chess and shogi, a Japanese version of chess), thus spawning the Alpha Zero system.

Despite the reduction in human-expert knowledge / experience in the Zero systems, they still had one important form of knowledge: the rules of the game. These rules are often called a **model**, and they incorporate not only what configurations and moves are legal, but also perfect information about the mapping from a state-action pair to a next-state-and-reward pair. These perfect models are standard for many board games but are less accessible for video arcade games, wherein complete and detailed simulations are necessary to determine the consequences of actions, such as the movement of a paddle, the tossing or shooting of projectiles, etc.

Successful AI systems for these arcade games often lack a complete model and are called *model free* systems. They can still learn a very skilled **policy** (i.e., mapping from states to actions or to probability distributions over actions), but they do so without ever learning **the** model. MuZero goes beyond those systems by learning **a** model, but one among abstract states (akin to hidden-layer activation patterns of a neural network). This new type of model provides useful scaffolding for learning a highly-skilled policy, but it never constitutes an

actual model of the real game. Amazingly enough, MuZero beats or matches Alpha Zero in chess, go and shogi and soundly beats the state-of-the-art systems in Atari video games. By removing more and more of the *biases* that might affect what and how an AI gamer learns, Google DeepMind has shown, once again, that *less is more*.

In this project, you will implement *the essence* of MuZero from scratch and then apply it to an arcade game of your choice. You will use an on-policy u-MCTS approach (i.e., the target policy and behavior policy are the same), with the policy generated by a trio of interlinked neural networks: the *Trinet* or Ψ in the lecture notes. The u-MCTS will provide action recommendations for the RL algorithm that runs each episode of the game, and, in turn, those episodes will supply target data for training Ψ. The interactions between these systems embody an extreme form of bootstrapping, as is common with RL systems, with Ψ determining actions in u-MCTS, which provides data for training Ψ. The full details of this process are explained in the lecture notes and videos.

This is a challenging assignment that requires a tight integration between four relatively complex programs: a game manager (i.e., a simulator for the video game), your u-MCTS and RL modules, and a software package for simulating neural networks such as JAX (recommended), PyTorch or Tensorflow.

Before beginning this project, your group should review all of the course lecture materials on RL, MCTS, JAX, and, of course, MuZero.

2 MuZero Overview

A reasonably complete explanation of MuZero resides in the lecture materials for this assignment, while a host of very specific details are found in the main publication from Google DeepMind:

Mastering Atari, Go, chess and shogi by planning with a learned model, Schrittwieser et. al., **Nature**, 2020.

This is available on the supplementary materials section of the course webpage.

Students should go through all of these sources before trying to understand the pseudocode below. This code provides a rough guideline to the main steps in MuZero, but your actual implementation should involve a collection of classes, subclasses and methods that cleanly separate:

1. Reinforcement Learning (RL) involved in generating episodes of game play, using policies to produce actions, recording rewards / penalties, and calculating state values based on bootstrapping, discounting, etc.,
2. video-game simulation,
3. generation of and movement within the u-MCTS search tree, and
4. building, training and deployment of MuZero's 3 neural networks.

Figure 1 sketches these modules and the basic relationships between them. If implemented properly, your system should require only minor modifications to handle many other arcade-style video games, as well as two-player board games.

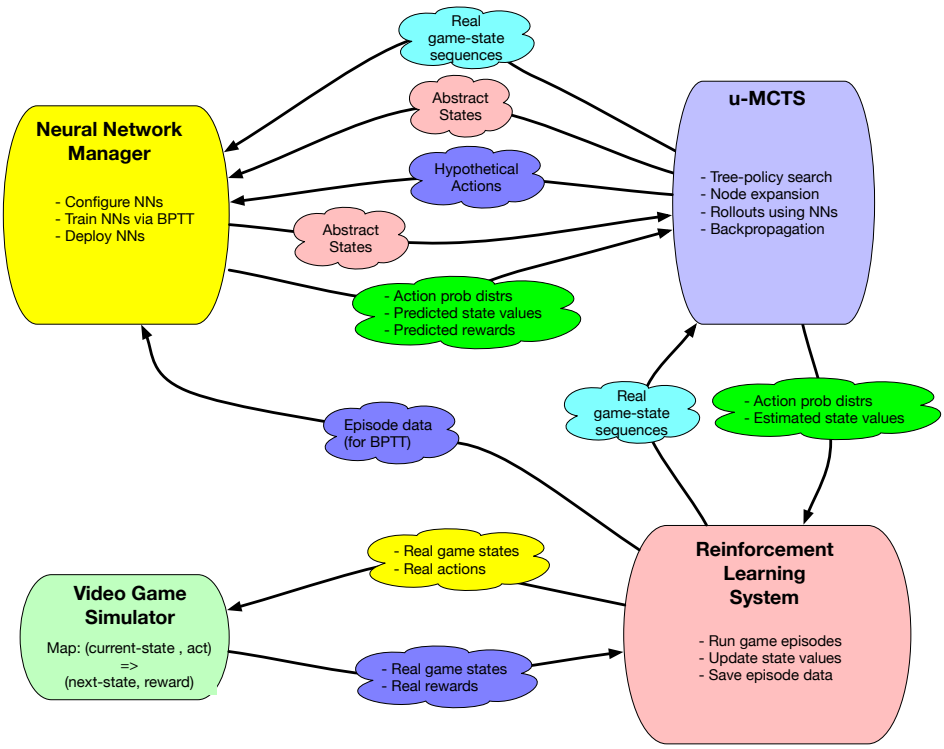

Figure 1: The main components of MuZero for learning to play video games.

2.1 The Basic MuZero Algorithm

MuZero uses a version of Monte Carlo Tree Search (MCTS), denoted *u-MCTS* in this document. The interplay between an episode-running RL system and u-MCTS enables MuZero to learn a policy for the game by gradually tuning the 3 interacting neural networks (Ψ). The **episode_loop** procedure (below) captures those basic interactions and outputs the final, trained, version of Ψ. Given Ψ, your system should remove the dynamics network (NN_d) and simply connect the representation and prediction networks (NN_r and NN_p, respectively) to form a trained policy network, a.k.a. *actor*. Your AI system can then play many more episodes of the video game and use the actor to choose moves when given the real-game state, but without further use of u-MCTS nor additional learning.

KEY PARAMETERS

- I_t := training interval for the three neural networks: $\Psi = (NN_r, NN_d, NN_p)$
- N_e := number of episodes; N_{es} := number of steps in each episode
- M_s := number of searches in the u-MCTS tree (u-Tree)
- d_{max} := maximum search depth in a u-Tree
- A := complete set of actions
- mbs = minibatch size

* Below, the FOR loops (of form *for i in range(n)*) are assumed to run from i = 0 to i = n-1.

EPISODE_LOOP()

1. EH $\leftarrow \emptyset$ # *Episode History = all training data from all episodes.*
2. Randomly initialize parameters (weights and biases) of Ψ
3. For episode in range(N_e):
   (a) Reset the video game to an initial state (s_0)
   (b) epidata $\leftarrow \emptyset$ # *episode data (used for training Ψ)*
   (c) For k in range(N_{es}) :
   - $\phi_k = \{s_{k-q}, \ldots, s_k\}$ # *Gather q+1 real-game states. Fill with blank states when $k < q$.*
   - $\sigma_k = NN_r(\phi_k)$ # *Create the abstract state*
   - Initialize u-Tree root with abstract state σ_k.
   - For m in range(M_s):
     – Use tree policy π_t to search from root to a leaf (L) at depth d of u-Tree.
     – $\forall a_i \in A$:
       ∗ Generate child node c_i with abstract state $\sigma_i = NN_d(\sigma_L, a_i).state$
       ∗ On the ith edge from L, attach the reward = $NN_d(\sigma_L, a_i).reward$
     – Choose a random child node (c^*) of L,
     – accum_reward = DO_ROLLOUT(c^*,$d_{max} - d$,NN_d,NN_p)
     – DO_BACKPROPAGATION(c^*,root, accum_reward)
   - π_k = normalized distribution of visit counts in u-Tree along all arcs emanating from root.
   - v_k^* = value of the root node
   - Sample action a_{k+1} from π_k
   - $s_{k+1}, r_{k+1}^* = Simulate\_game\_one\_timestep(s_k, a_{k+1})$ # *Get new state and reward*
   - epidata.append($[s_k, v_k^*, \pi_k, a_{k+1}, r_{k+1}^*]$)
   (d) EH.append(epidata)
   (e) if episode modulo $I_t == 0$:
   - DO_BPTT_TRAINING(Ψ,EH,mbs)
4. Return Ψ # *The fully-trained neural networks: $NN_r, NN_d, and NN_p$*

DO_ROLLOUT(node,depth,NN_d,NN_p):

- σ = node.state
- accum_reward $\leftarrow \emptyset$
- For d in range(depth):
  - $\pi, v = NN_p(\sigma)$ # *Use neural net to predict policy and value for* σ
  - Sample action a from π
  - $\sigma, r = NN_d(\sigma, a)$
  - accum_reward.append(r)
- $\pi, v = NN_p(\sigma)$
- accum_reward.append(v)
- Return accum_reward

DO_BACKPROPAGATION(node,goal_node,rewards)

- node.visit_count += 1
- node.update_Q_value(sum(rewards)) # *summing of rewards may include discounting*
- if node $\neq$ goal_node:
  - e $\leftarrow$ edge from node.parent to node
  - DO_BACKPROPAGATION(node.parent,goal_node,rewards.append(e.reward))

DO_BPTT_TRAINING(Ψ, episode_history,mbs):

- q = look-back; w = roll-ahead
- for m in range(mbs):
  - Choose a random episode (E_b) and state ($s_{b,k}$) from episode_history
  - Gather training data from E_b:
    - States: $S_{b,k} = \{s_{b,k-q}, \ldots, s_{b,k}\}$ from E_b
    - Actions: $A_{b,k} = \{a_{b,k+1}, \ldots, a_{b,k+w}\}$
    - Policies: $\Pi_{b,k} = \{\pi_{b,k}, \ldots, \pi_{b,k+w}\}$
    - Values: $V^*_{b,k} = \{v^*_{b,k}, \ldots, v^*_{b,k+w}\}$
    - Rewards: $R^*_{b,k} = \{r^*_{b,k+1}, \ldots, r^*_{b,k+w}\}$
  - Do BPTT using Ψ, $S_{b,k}$, $A_{b,k}$ and $PVR^* = [\Pi_{b,k}, V^*_{b,k}, R^*_{b,k}]$ as shown in Figure 2
- Return Ψ

Exact details of the backpropagation through time (BPTT) variant needed to train Ψ are not included in this document. Although this is covered in greater depth in the lecture notes, it is purposely left as an exercise for students to work out all of the intricasies of the algorithm. Remember that JAX takes care of the most difficult aspects of BPTT: computing gradients. So although a review of gradient calculations in BPTT will help your own understanding of the process (and appreciation for the heavy lifting done by automatic differentiation), it is not essential for successfully completing this assignment.[1]

[1]For a lengthy video overview of BPTT for Elman networks, go to the *Materials* section of the course web page and follow the *Archive of Deep-Learning course (IT-3030)* link. There, you will find 4 lectures on BPTT in week 10.

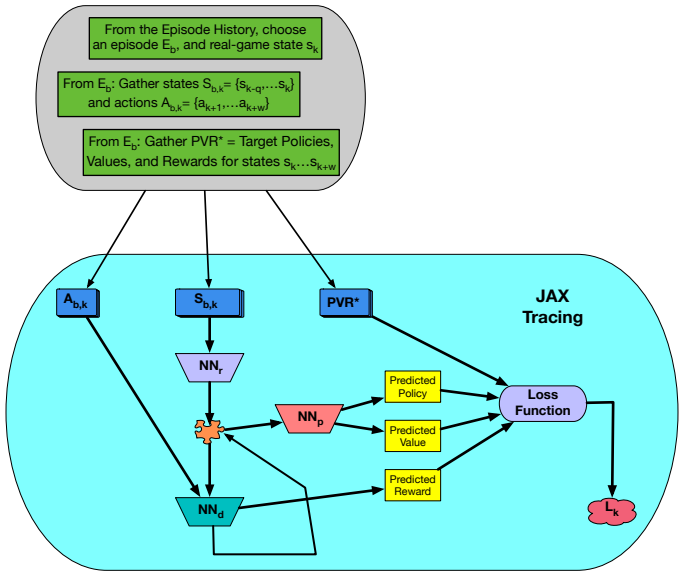

Figure 2: Overview of the training of MuZero's three neural networks via backpropagation through time (BPTT).

2.2 MuZero Classes

The following are **suggested** classes and a few of their most important methods for this assignment. You are free to choose other classes for your object-oriented implementation. In general, there will be a lot of interaction between the objects of this assignment. MuZero exhibits a very high level of bootstrapping, so this often entails a lot of data exchange between subsystems that do lookahead / plan (u-MCTS), those that play actual episodes of the game, and those that learn.

2.2.1 Video Game Simulator

This maps the pair (game state, action) to the pair (new game state, reward). For arcade games, these may require physical simulations of movement, acceleration, collision, etc. and thus require a good deal of computation.

2.2.2 Game State Manager (GSM)

This understands the game and how the combination of a state and action produces a new state, and possibly a reward. It can also recognize final states (for games that actually have goal states). It may use a video-game simulator to perform mappings, but then it can cache the results of these mapping such that later calls to the state manager can involve a simple lookup instead of a run of the game simulator, which can be much more computationally demanding.

Typical methods for a state manager are therefore:

- Generate the initial state for a game.

- Given a game state, return all legal actions that can be performed in that state.
- Given a game state, return one good action to perform, or a probability distribution over the actions to perform. Some GSM implementations may employ a neural network to do this mapping, while others use a lookup table.
- Given a game state, determine whether it is a final / goal state or not.
- Given a game state, return an evaluation of that state. In some implementations, the GSM may use a neural network to perform this evaluation.
- Given a game state and an action, return the next state and reward. For non-deterministic games, this may return a set of possible next states and rewards.

For classic Monte Carlo Tree Search, the GSM is useful in expanding MC tree leaves, since it can take the game state in a leaf, generate all legal actions, and then all states (and rewards) resulting from those actions. Thus, the GSM can help MCTS do a full expansion of a parent tree node, i.e. produce all child nodes and labels (action, reward) on the tree edge from parent to child.

2.2.3 Abstract State Manager (ASM)

This performs some of the same functions as a GSM, but since abstract states do not directly reflect game states, the method set is restricted. Typical methods will:

- Map a sequence of game states to a single abstract state. This involves the representation neural network, NN_r.
- Given an abstract state, return all legal actions. This is normally all possible actions, except for abstract states used as the root of a u-MCTS tree, which do reflect a particular game-state sequence and thus a particular current state (i.e. the last state in the sequence).
- Given an abstract state, produce a policy (i.e. probability distribution over actions) and a predicted value. This employs the prediction network, NN_p.
- Given an abstract state and action, return the next abstract state and a predicted reward. This uses the dynamics network, NN_d.

2.2.4 Neural Network (NN)

You may implement individual neural networks in various ways, but it is usually good to have one class to wrap around whatever implementation (e.g. JAX, PyTorch, KERAS) that you choose. Typical methods include:

- Building the network from a configuration file that specifies details such as the number, size and type of layers, plus their interconnectivity.
- Preprocessing inputs to the network.
- Running the network in forward mode: mapping inputs to outputs.
- Postprocessing the outputs of the network.

- Training the network - Although this is normally a general operation for individual networks and is thus a generic method, MuZero involves a more complex training procedure that involves the combination of 3 networks, so training is best to perform in the Neural Network Manager (NNM) class (see below).
- Saving the parameters (weights and biases) of a trained network to a file.
- Loading the parameters of a trained network from a file.

2.2.5 Neural Network Manager (NNM)

This keeps track of the 3 core neural networks used in MuZero. It may serve as a distributor of tasks such that the proper network gets called for each of the three main operations: representation, dynamics and prediction.

The NNM's most important method is training, since NN learning in MuZero involves backpropagation through time (BPTT) in a composite network (housing all 3 basic networks), as explained in the lecture notes.

2.2.6 MuZero Monte Carlo Tree Search (u-MCTS)

This performs all of the standard operations of MCTS, but using abstract states. As such, it operates very similarly to the standard MCTS, which is explained in separate lecture materials for this course. Typical methods perform operations such as:

- Using the tree policy to search to a leaf node.
- Expanding a leaf node.
- Performing a rollout from a child of the expanded leaf node.
- Backpropagating final-state (S) evaluations back to the u-MCTS tree root (R).
- Updating visit counts and Q values along the path from S to R.
- Returning a probability distribution over actions and an evaluation (of the root state) after many traversals of the tree. Remember: the root abstract state **does** correspond to a game state, so evaluations of the root can be associated with that game state.

This uses NN_r to generate the initial abstract root state, NN_d to create child states (and predicted rewards), and NN_r to produce policy and state-evaluation predictions. Some or all of these operations might be done via calls to the abstract-state manager (ASM) or the NNM, depending upon how you structure your system.

2.2.7 Episode Buffer (EB)

This keeps track of the episodic data that will be used to train the composite neural network. Each episode (E) will consist of:

1. The sequence of game (not abstract) states for the entire episode (E).

2. The sequence of actions performed in E.
3. The sequence of rewards achieved in E.
4. The sequence of policies generated by u-MCTS during E.
5. The sequence of game-state evaluations from the root of the u-MCTS tree.

The reinforcement learning manager (see below) and the u-MCTS will supply these 5 aspects of an episode.

When training the composite neural network, the NNM can call the EB and request a portion of any episode, which it then uses as a "minibatch" for training the coupled networks, with game states and actions used as inputs, while policies, rewards and state evaluations serve as targets (as described in the lecture notes).

Thus, along with saving new episodes into the buffer, the EB can fetch chunks of state-action-reward-policy-evaluation tuples from any episode.

2.2.8 Reinforcement Learning Manager (RLM)

Reinforcement Learning (RL) is being performed by several of the above classes, but it might still be useful to have a class that handles relatively high-level operations such as:

- Resetting the game to a start state (that it probably fetches from the state manager (SM))
- Running episodes of the game, with help from SM and u-MCTS.
- Sending short sequences of game states to the ASM to get an abstract state that u-MCTS can place in its root node.
- Calling u-MCTS to get probability distributions over actions in order to make the next move in the game.
- Keeping track of the game states, actions, and rewards produced, along with the data (policies and evaluations) returned by u-MCTS.
- Off-loading the above information to the episode buffer (EB).

Note that the RLM works almost exclusively with game states, except when (if) used to supply u-MCTS with a starting abstract state that it gets from the ASM.

3 Simple Video Games

You are free to use any arcade game of your choice., but it must have a gridded input, whether actual pixels or some higher-level representation. Here are some sites to explore for interesting RL environments:

1. A fork of *OpenAI Gym* called *Gymnasium*: https://gymnasium.farama.org/

2. More complex environments (many 3d) by Unity Technologies: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Examples.md

3. A comprehensive overview of RL environments : https://github.com/clvrai/awesome-rl-envs

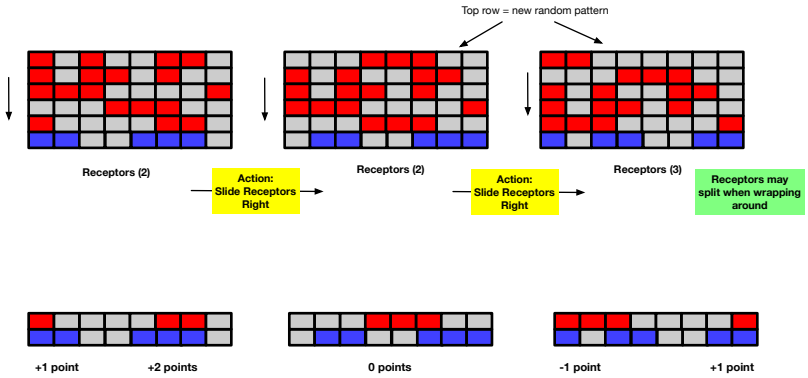

Figure 3: The game of BitFall. Debris (red segments) falls straight down, one row per timestep. The receptors (blue segments) can only be moved as a complete row, one column left or right (or remain stationary) on each timestep. A comparison of the receptors to the bottom row of debris yields intermediate rewards at the end of each timestep. P positive points result when a receptor segment completely overlaps a debris segment with excess on one or both sides; P is the size of the debris segment. N negative points occur when a debris segment completely overlaps a receptor with excess on one or both sides; N is the size of the receptor segment. When a receptor segment of size X moves past the left or right edge, it splits into two receptors of size 1 and X-1. Another shift in the same direction yields receptors of size 2 and X-2. Reversing these shifts returns the receptor to its original size.

Your chosen game need not be complex; you may even want to develop it yourself, to have full control of the code. It is highly recommended to choose a game with a scaleable grid so that you can debug your system with very small and simple grids before advancing to larger arrays. As an example of such a simple, scaleable game, Figure 3 portrays BitFall (whose inventor surely deserves an international gaming award). The rules are simple: the red bit segments (a.k.a. debris) fall down at the rate of one row per timestep, while the bottom row of receptor segments (blue) can remain static or be shifted one cell left or right on each timestep. The interaction between the lowest row of debris and the receptors determines the intermediate rewards / penalties; it is similar to criteria used in simple battle games, where out-numbering the opponent incurs a win, being outnumber incurs a loss, and ties yield nothing. In BitFall, two segments are compared. If one segment a) is larger than the other, b) completely covers the other and c) has overlap on one or both sides, then it *wins* and the number of points achieved equals the size of the losing segment. A receptor win entails positive points, while a debris win yields negative points.

A random row of debris bits is added to the grid at each timestep, while the bottom debris row is removed, after it is compared to the receptors for point generation. The original pattern of receptors is chosen by the user, but it can change during the game if a segment shifts beyond the left or right edge and thus splits, creating two segments that the game treats independently when tallying points. BitFall has no *final state*; it is a continuous game. So the user just plays it for some predetermined number of timesteps and tries to accumulate as many points as possible during that period.

BitFall is quite trivial to implement and can scale to almost any size of grid. Furthermore, the restricted set of moves yields a very low branching factor for Monte Carlo Tree Search. Many similar alternatives surely exist. There is no need to get carried away with Donkey Kong, Pac-Man or some complex shooter game for this assignment. When paired with MuZero, even something as simple as Break-Out might be too much for most computers.

This course does not provide any special server access to students, so focus on games that fall within your own computational resources.

4 The Educational Video

Your group will produce one 10-minute video that explains MuZero and illustrates its use with a single-player, Atari-style video game. The goal is to convey as much important technical information as possible in the most understandable manner, and all in 10 minutes or less. The key themes in the video must include reinforcement learning (RL), model-based -vs- model-free RL, Monte Carlo Tree Search (MCTS) and, of course, MuZero's relationships to all of them. You must also introduce the video game that your group has tackled along with the key details of your system and the results it produces.

In short, the video **must**:

1. Introduce **all** important AI concepts, including RL, model-based -vs- model-free, MCTS and MuZero.
2. Explain (very briefly) the video game,
3. Give as much detail as possible about your group's implementation of MuZero, and
4. Show the results of your system when applied to the video game.

4.1 The Audience

You should assume that the audience for this video is students who have already learned the basics of AI through courses such as TDT-4136 (AI Introduction), TDT-4172 (ML Introduction), and TDT-4171 (AI Methods), but you should not assume that they know much (if anything) about RL, MCTS, AlphaGo, AlphaZero or MuZero. You should definitely **not** waste time explaining the basics of neural networks, tree search or any other standard AI tools.

4.2 Diagrams

It is **strongly** recommended that the majority of your visual material is in pictoral form. Images often convey more useful information, more efficiently, than text; and this becomes very important with the 10-minute time restriction. You may decide to draw these images freehand on an online whiteboard, or you may choose more conventional diagramming tools, or a combination. Any of these approaches can serve your purpose. Many educational blogs and vlogs get excellent results using freehand whiteboard sessions with a few accompanying PDF diagrams.

A diagrammatic overview of your code, possibly with a few snippets of pseudocode, is much preferable to the actual code details, particularly under these strict time constraints. Your goal should not be to allow the viewer to exactly reproduce your system and results, but to convey a basic (yet still technical) understanding of what you have done.

It is explicitly **forbidden** to use diagrams that are not of your own design and production in the video. The forbidden diagrams include anything found in the materials for this course: lecture notes, research articles,

youtube videos, etc. Violation of this rule will incur a score of ZERO for the project. Copying of images is a matter that is taken very seriously in the media – in many cases it cannot be done without formal permission from the original source. In this project, one main part of your challenge is to produce diagrams that deliver your own message in the clearest and simplest form. We are interested in how **your group**, not Google DeepMind, explains all aspects this project.

4.3 Animations

Simple, abstract animations are often excellent tools for explaining complex concepts, so you should definitely consider using them. To generate animations in matplotlib, you might want to use *FuncAnimation*, as nicely described in this blog post by Ken Hughes:

https://brushingupscience.com/2016/06/21/matplotlib-animations-the-easy-way/

Of course, there are many other free animation tools available online.

4.4 Humans in the Loop

It is **not** important that every group member speaks in the video. Too many voices could easily detract from the message. So think carefully about how many to use. We assume that all group members make significant contributions to the project. They don't all have to appear in prime time.

Video of the speaker's face or that of other group members should be limited or completely avoided. The 10-minute time restriction entails that anything shown on the screen should be providing important information; a *talking head* might be the best way to convey certain types of information, but most of the required content is quite technical and probably best transmitted via diagrams, equations and short animations (e.g. of the video game in action, of search progress in an MCTS tree, etc.).

4.5 Movie Contents

Your movie must not exceed 10 minutes in duration. Violations of this constraint will incur the loss of one, maybe two points. A key challenge of this project is to concisely and accurately present your case, and not to ramble on and on (as your instructor often does during lectures).

Your movie **must** include the following:

1. A short, but informative, explanation of the key concepts embodied in MuZero (1 point)
2. As detailed as possible a description of your group's implementation of MuZero. (2 points)
3. A quick overview of the chosen video game. (0 points if it's there, but -1 point if not)
4. A display and description of the results of running your system on the video game (1 point)

5. A **credits** page listing the full names of all team members and a few buzzwords about their main contributions. This page should not be on the screen for more than a few seconds, but we need it for doing our own *credit assignment* to all members of your team. Everyone will get the same number of points.

5 Deliverables

The video is the only deliverable. It will be worth FOUR points (of the 6 total points available for this class). If all of the items listed under *Movie Contents* are covered with *reasonable quality*, your group should get all 4 points. However, failure to cover any items, extreme sloppiness, or a duration over 10 minutes can easily cost a point (or two). The actual quality of your results is not so important. Just make sure that they are clearly shown and briefly discussed. Regardless of whether the results are good or bad, do your best to explain WHY they ended up that way.

There is no written report, nor demonstration for this project. Simply upload a **zipped** version of your movie to BLACKBOARD by NOON on the delivery date. In the text associated with your file, include the names of every group member. Only ONE group member should upload the video.