# Programming Assignment 3: Tetris

Due: Thursday 05/01/2025 @ 11:59pm EST

The purpose of programming assignments is to use the concepts that we learn in class to solve an actual real-world task. We will not be using Sepia for this assignment: I have developed a game engine for us to use. In this assignment we will be writing agents to play Tetris.

**WARNING:** There are **NO EXTENSIONS** for this assignment. This assignment has a **firm** due date of the last day of class.

## Reinforcement Learning via Sampling

Reinforcement Learning is a style of artificial intelligence where we aren't given input data and we aren't given ground truth. Instead, we are given a world in which our agent will act. Our agent, in order to learn, must either employ supervised or unsupervised learning techniques, but what is the data to learn from? In Reinforcement Learning, we collect data by acting in the world. To do this, we must first engineer the senses our agent has so that our agent can perceive the state of the world. We then must engineer the way our agent makes decisions so that it can generate actions when in certain states. Initially, these decision will seem random, and that's ok, we haven't learned anything yet. In order to learn, our agent must have access to something called a *reward function* so that it can be rewarded when its decisions lead to "favorable" states, and punished when those decisions do not lead to favorable states. Sometimes the reward function is given to us, and sometimes we have to engineer it ourselves. In this scenario, you are not given the reward function and must engineer it yourself (smaller reward values correlate to harsher punishments).

With access to states and a reward function, we can employ supervised learning techniques to learn by acting in the world. The goal here is to learn a **policy**: a function that maps states to the (optimal) action that should be taken. We will do this with the following paradigm:

1. Play a bunch of training games. In these games, the agent is "frozen" (i.e. unable to learn anything). The point of these games are to generate training data in the form of (state, reward, next_state) triples. What is important during this step is that we don't listen to our model every time. There is some small chance that we ignore what our model thinks is the optimal action to do and instead chooses to explore some other action.
2. Once we have generated training data, convert this training data into a supervised learning dataset. How we do this depends on what our model is trying to learn, which we'll talk more about in the next section. However we generate ground truth values, once we have this supervised learning dataset, we train our model a little bit on the dataset.
3. After our model is trained, we want to see how well it performs. So we play a bunch of evaluation games where the model is again frozen, but we will always listen to it (unlike in the training games where we could sometimes ignore the model).

Solving a Reinforcement learning problem like this is nice because it is guaranteed to learn the optimal model in the limit of infinite data. The downside to solving the problem like this is because of the practicality of "learning in the limit of infinite data." Sure, the model is guaranteed to get better eventually, the question is practically how long does it take? Training a model like this can take **ages**, its up to the quality of our engineering decisions and luck!

## Reinforcement Learning with Q-learning

Q-learning, as we will see in lecture, is a specific flavor of Reinforcement learning. In Q-learning, the model learns whats called a *Q-function*, which guesses how good an action is in a state $Q(s, a)$. If our model was optimal, then whenever our agent is in a state, we could calculate the best action to do via

$$a^* = \underset{a \in Actions(s)}{\arg\max}\ Q(s, a)$$

Q-learning tells us how to convert the training data generated from samples into a supervised learning dataset. Most of the work occurs in generating the ground truth. If we have a bunch of samples $(s, a, s')$ from playing the game ($s$ was a state, $a$ was the action taken in state $s$, and $s'$ was the resulting state of applying $a$ in $s$), then we generate ground truth using whats called the *Bellman* equation (for Q-functions):

$$Q(s, a) = R(s, a) + \gamma \max_{a' \in Actions(s')} Q(s', a')$$

We'll talk about this in lecture a lot once we get to Q-functions. So, with this equation, we convert the training data into a supervised learning dataset, and we can then train our model.

## Reinforcement Learning with Neural Networks

Reinforcement learning requires us to learn, in the case of q-learning, a single Q-value per (state, action) pair. This mapping is not feasible to store as a table for all but the simplest toy problems. For most problems (Tetris included), there are simply too many states for us to allocate a single float per entry. When we encounter a scenario like this, we often use a neural network in place of the tabular function. Neural networks learn functions, so we can use them in this setting to learn the function while skipping the memory requirements for an exhaustive table.

Like all ML models, neural networks learn a kernel function $K(\vec{x}, \vec{y})$: a similarity function. This kernel is both a blessing and a curse. A kernel is a blessing because, if you see an example (that you've never seen before), you can leverage the knowledge that you've learned from similar examples (the kernel tells you how similar things are). A Kernel is a curse because of how it is constructed. A Neural Network learns a kernel through its parameters. If we change a single parameter (i.e. tweak one of the many parameters inside the network), we change the **entire** kernel and therefore we change how the network views **every** example it will ever see. This means that if we try to improve performance on a single example, we can potentially destroy the knowledge that the network currently has contained within it. It is for this reason that when we train neural networks, we don't optimize for a single example: we take the average gradient across a batch of examples.

While we technically should update our network as soon as a transition occurs, we will instead aggregate them in a buffer called a *replay buffer*. The purpose of the replay buffer is to account for the curse of the kernel: if we updated our network with a transition the second we observe it, we would be optimizing for a single example and run the risk of our network forgetting all the knowledge it has already learned. So, instead of updating the network as soon as transitions occur, we will collect them into the replay buffer, and wait until the network has finished playing a bunch of games before using the replay buffer to update the network. To be clear, a replay buffer is a hack: this is a bandaid solution that does not solve the underlying problem. A more meaningful approach would be to address neural networks treating each example as i.i.d, but thats a topic for another time and is an active area of research.

Therefore, to be more accurate, we should update the training paradigm from the earlier sections. Training is organized into cycles. Each cycle has the following sections, which are executed in order:

1. Play a bunch of training games, the sole purpose of which is to observe (and record) trajectories to populate the replay buffer.
2. Convert the replay buffer into a supervised learning dataset (using our Bellman equation from earlier to calculate ground truth). Train the neural network on this dataset.
3. Play a bunch of evaluation games with the updated neural network, the sole purpose of which is to evaluate the performance of the network.

This cycle is repeated an enormous amount of times. Eventually, the network will get really good, and therefore our agent will get really good.

**Tetris**

The version of tetris in this assignment is slightly different than how humans play tetris. When a human plays tetris, they use a joystick (or arrow keys) to guide a piece as it descends to control where the piece lands on the board. While we could do this same process for our agent, it would take much longer for the agent to learn. Instead, the way our agent plays tetris is it gets to choose the "final resting place" of a piece, and then once that decision is made, the piece teleports to the desired location. The reason is that the machine does not have the same dexterity that we humans do: we naturally are pretty good at guiding a piece as it descends to the desired location. If we were to have the machine also guide the pieces, then we would be asking the agent to learn multiple things: it would need to learn *where* the piece should go, and it would also need to learn *how* to get it there. This is a much harder problem that would take much longer to learn, so the machine gets assistance in placing pieces where it wants them to go.

The rest of tetris is still the same. You earn points for clearning lines: the more lines you clear at once the more points you earn. A *tetris* is clearning four lines at once, and you get bonus points for doing so. A *perfect clear* is where you clear the board, and you get extra points for doing so.

This version of tetris also implements single and double *t-spins*. A "T"-piece (also called a "T"-block or a "T"-mino) can be squeezed into locations that it normally doesn't fit by rotating the piece at the correct time. If you are able to squeeze a "T"-piece using a single rotation, that is called a *single* t-spin. If you are able to squeeze a "T"-piece using two rotations, that is called a *double* t-spin. You are rewarded extra points for a single t-spin, and even more points for a double t-spin.

## 1. Copy Files

Please, copy the files from the downloaded lab directory to your cs440 directory. You can just drag and drop them in your file explorer.

- Copy `Downloads/tetris/lib/tetris-<VERSION>.jar` to `cs440/lib/tetris-<VERSION>.jar`. This file is the custom jarfile that I created for you.
- Copy `Downloads/tetris/lib/argparse4j-0.9.0.jar` to `cs440/lib/argparse4j-0.9.0.jar`. This is a jarfile that `tetris.jar` depends on. It provides similar functionality to Python's `argparse` module. The documentation for `argparse4j` can be found here.
- Copy `Downloads/tetris/src` to `cs440/src`. This directory contains our source code `.java` files.

- Copy `Downloads/tetris/tetris.srcs` to `cs440/tetris.srcs`.
  This file contains the paths to the `.java` files we are working with in this assignment. Just like in the past, files like these are used to speed up the compilation process by preventing you from listing all source files you want to compile manually.
- Copy `Downloads/tetris/doc/pas` to `cs440/doc/pas`. This is the documentation generated from `tetris.jar` and will be extremely useful in this assignment. After copying, if you double-click on `cs440/doc/pas/tetris/index.html`, the documentation should open in your browser.
- Copy `Downloads/tetris/learning_curve.py` to `cs440/`. This is a python script that will be useful for plotting the performance of your agent as a function of how many games its played. More on this later in this document.

**2. Test run**

If your setup is correct, you should be able to compile and execute the following code. A window should appear:

```
# Mac, Linux. Run from the cs440 directory.
javac -cp "./lib/*:." @tetris.srcs
java -cp "./lib/*:." edu.bu.tetris.Main

# Windows. Run from the cs440 directory.
javac -cp "./lib/*;." @tetris.srcs
java -cp "./lib/*;." edu.bu.tetris.Main
```

**NOTE:** There are several command line options available to you. If you want to see the exhaustive list, please add a `-h` or `--help` argument to the command line and see the help message! Most of these command line arguments are your way of configuring the way you train your model (such as learning rate, batch size, etc.).

**Task 1: `TetrisQAgent.java` (100 points)**

Please complete the implementation of `TetrisQAgent.java`. In this file, I am asking you to devise and implement several different methods which are crucial for a good RL agent. I have listed them below in order of priority:

1. method `getQFunctionInput`. This method takes a `GameView` object which contains all the information about the current state of the game (i.e. the *state*), as well as a `Mino` object. A `Mino` (short for `Tetramino`) are the pieces that you can place in a game of tetris. The `Mino` provided here as an argument is a possible resting place for the `Mino` that needs to be placed on the board (i.e. the *action*).

   Your method needs to convert these two objects into a row-vector which will be used as input to your neural network. You are responsible for engineering the representation, i.e. for engineering the conversion of game information into a vector. You want your vector to contain all the information necessary for the neural network to provide a meaningful ranking (q-value) of the `Mino` placed on the board.
2. method `getReward`. You have to engineer your reward function. This method takes a single `GameView` object as input, and your method should calculate the reward for being in that state of the game. If this state is bad, you should produce a small number (you are allowed to go negative), and if this state is good, you should produce a large number. I encourage you to be

creative when devising your reward function, we want to determine "goodness" and "badness" that correlates to actual good tetris behavior. Don't forget that you are responsible for assigning a reward for terminal states as well as a reward for nonterminal states. Don't be surprised if the way you measure "goodness" and "badness" is very similar to the features you engineer into your feature vector.

3. method `initQFunction`. Once you have designed your vector representation and figured out your reward function, you will now need to actually build your neural network object. You are only allowed to use feed-forward neural networks in this project: I could not get convolutions working in time for use this semester. The size of the input vector is now fixed: you had to decide this when implementing `getQFunctionInput`, so your input layer should expect a vector of that size. The output of your neural network should be an unbounded (i.e. no output layer activation function) scalar value (i.e. the q-value).

   The difficulty part will be how many hidden layers do you use, how big are each hidden layer, and what are their activation functions? I have implemented a little library of layers for you to take a look at, just know that convolutions are not ready yet, so please don't use them.

4. methods `shouldExplore` and `getExplorationMove`. These methods are how we implement curiosity into our agent. As your agent learns, it will start to discover actions that it thinks are good, and start suggesting them more frequently. This can be a blessing and a curse. The blessing is that the model is doing things that are "good" (as measured by the reward function). The curse is that as we continue to follow the policy, we will stop exploring and gaining novel experiences, and we risk getting stuck. This isn't really a problem if our policy happens to stumble upon really good actions, however this is unlikely, and instead our policy will get stuck on (what it deems to be the best its seen so far but are actually) mid-tier actions.

   To encourage the agent to ignore the policy (sometimes) and instead explore for novel experiences, we need to implement a version of curiosity. The first method, `shouldExplore` should return `true` when the agent determines that it should explore in this state (and ignore what the policy recommends), and `false` otherwise. The second method, `getExplorationMove` is how we generate an action that will (hopefully) lead to a novel experience (assuming we have decided to ignore the policy in the first place). You should implement some notion of curiosity here.

To earn full credit, you must demonstrate that your agent has learned. If your agent can score an average of 20 points when playing 500 games of tetris, then I will award you full credit.

## Training

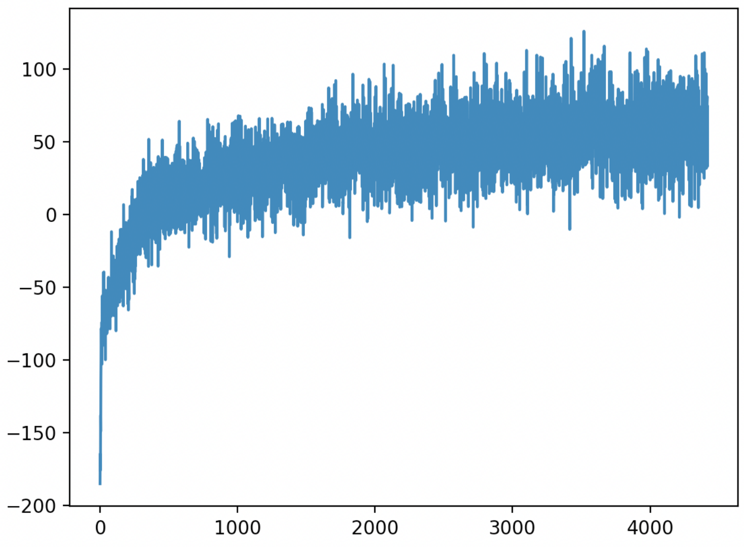
To train your agent, you should develop your code locally and make sure that it doesn't error, and that it measures quantities the way you intend them to be measured. Once you have that working, you will want to train your agent on tens of thousands, if not hundreds of thousands, if not millions of games. To do this, we are giving you access to the SCC. We will provide you a tutorial of how to use the SCC in lecture.

## Checking on Performance

When training your agent (which will take days), you will want to check in on it periodically. My code will, after every cycle, save your neural network to disk in a directory called `params`. You are free to change the destination of the saved models, and on the SCC I encourage you to save your models to the `projectnb` directory. When should you stop training, and which version of your network is the best? These are questions answered by the evaluation section of every cycle. The average trajectory utility is printed out after every cycle: this is information you will want to keep and to plot. To do so, when it is time for you to launch your agent's massive training run, you should run it like so (only shown for linux since I am assuming you'll be running this on a linux machine):

```
java -cp "./lib/*:." edu.bu.tetris.Main <OTHER_ARGS> | tee my_logfile.log
```

The `| tee my_logfile.log` part of this command will pipe stdout to a file called `my_logfile.log` which will exist in the same directory that the shell is in when the command was run (again I would recommend specifying a path to the `projectnb` directory). Periodically, you can download your `my_logfile.log` to your laptop, and then use `learning_curve.py` to plot the average trajectory utility as a function of time. The way you will know your model is learning is if you get something that looks like this



The x-axis is the number of cycles that the model has run for (this curve shows 4000 cycles), and the y-axis is the average trajectory utility.

## Task 2: What to Submit

This time you will need to submit more than just your Java source code. As your model trains, my code will write the parameters of the model to files after every cycle. You need to pick one of these files (ideally the file that corresponds to the model that has the best performance), rename it to `params.model`, and submit it along with your `TetrisQAgent.java` file on gradescope. Reminder that this filename is case sensitive and must be **exactly** `params.model` for the autograder to recognize it.

## Task 3: Extra Credit (50 points)

If your agent can earn an average of 40 points when playing 500 games of tetris, then I will award you full credit.

**Task 4: Tournament Eligibility**

In order for your submission to be eligible in the tournament, your submission must satisfy all of the following requirements:

- Your submission must be on time.
- You do not get an extension for this assignment.
- Your agent compiles on the autograder.
- Your agent can play 1000 games of tetris and earn an average of 20 points or more.