Importing Packages

```
using ReinforcementLearning
```

```
begin
    using Flux
    using Flux.Losses
end
```

```
using Random
```

```
using Intervals
```

```
using StableRNGs
```

```
using Plots
```

Defining Some Methods and Constants

```
duplicate (generic function with 2 methods)
```

```
begin
    duplicate(arr::Array) = vcat(arr, arr)
    duplicate(arrs...) = duplicate(vcat(arrs...))
end
```

```
moving_average (generic function with 1 method)
```

```
moving_average(arr::Array, n::Int) = [sum(arr[i:i+n])/n for i in 1:length(arr)-n]
```

`consts` initialized

```
begin
    const N_STEPS = 100000
    const NAMES = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]
    const VALUES_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
    const VALUES_2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10]
    const VALUES_3 = [14, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
    const SUITS = ["diamonds", "clubs", "hearts", "spades"]
    const SUITS_TO_COLORS = Dict(
        "diamonds"=>"red",
        "clubs"=>"black",
        "hearts"=>"red",
        "spades"=>"black"
    )
    const UNO_COLORS = ["blue", "green", "red", "yellow"]
    const UNO_VALUES = vcat([0], duplicate(Array(1:9)))
    const UNO_SPECIALS = duplicate(["+2", "reverse", "skip"])
    const UNO_WILDS = duplicate(duplicate(["wild", "wild+4"]))
    md"`consts` initialized"
end
```

```
const DeckTypes =
  Dict(:StandardNoJokers => [Card("A", 1, "diamonds", nothing), Card("A", 1, "clubs", nothin
```

```
const DeckTypes = Dict(
    :StandardNoJokers => [
        Card(NAMES[i], VALUES_1[i], SUITS[j])
        for i in 1:13
        for j in 1:4
    ],
    :StandardWithJokers => vcat(
        [
            Card(NAMES[i], VALUES_1[i], SUITS[j])
            for i in 1:13
            for j in 1:4
        ],
        [
            Card("Joker", 0, nothing, "wild"),
            Card("Joker", 0, nothing, "wild")
        ]
    ),
    :StandardNoJokersFace10 => [
        Card(NAMES[i], VALUES_2[i], SUITS[j])
        for i in 1:13
        for j in 1:4
    ],
    :StandardNoJokersAceHighest => [
        Card(NAMES[i], VALUES_3[i], SUITS[j])
        for i in 1:13
        for j in 1:4
    ],
    :DoubleStandardNoJokers => duplicate(
        [
            Card(NAMES[i], VALUES_1[i], SUITS[j])
            for i in 1:13
            for j in 1:4
        ]
    ),
    :DoubleStandardWithJokers => duplicate(
        [
            Card(NAMES[i], VALUES_1[i], SUITS[j])
            for i in 1:13
            for j in 1:4
        ],
        [
            Card("Joker", 0, nothing, "wild"),
            Card("Joker", 0, nothing, "wild")
        ]
    ),
    :StandardUno => vcat(
        [
            Card(string(UNO_VALUES[i]), UNO_VALUES[i], UNO_COLORS[j])
            for i in 1:19
            for j in 1:4
        ],
```

```
        [
            Card(UNO_SPECIALS[i], UNO_COLORS[j], UNO_SPECIALS[i])
            for i in 1:6
            for j in 1:4
        ],
        [
            Card(UNO_WILDS[i], "black", UNO_WILDS[i])
            for i in 1:8
        ]
    )
)
```

Defining Structs

```
mutable struct Card
```

```
Card(name::String, value::Int64, suit::Union{Nothing,String}) = new(
```

```
mutable struct Deck
```

```
mutable struct Player
```

```
mutable struct NamedPlayer
```

Defining Specific Methods

```
broadcastDict (generic function with 1 method)
```

```
broadcastDict(dict::Dict, arr::Array) = map(x->dict[x], arr)
```

valuesOnly (generic function with 2 methods)

- begin

sumHand (generic function with 2 methods)

- begin

isBust (generic function with 2 methods)

- begin

shuffleDeck (generic function with 1 method)

- shuffleDeck(d::Deck) = begin

drawCard (generic function with 2 methods)

- begin

nthPlayedCard (generic function with 1 method)

- begin

removeCardFromPlayer (generic function with 2 methods)

- begin

playCard (generic function with 2 methods)

- begin

addCardToBottom (generic function with 2 methods)

- begin

addDrawpileToHand (generic function with 2 methods)

- begin

isBook (generic function with 2 methods)

- begin

getCounts (generic function with 1 method)

- `getCounts(hand::Array{Card,1}) = [sum((x->x.value==i).(hand)) for i in 1:13]`

- `Base.convert(::Type{Int}, ::ReinforcementLearningCore.NoOp) = 0`

skipPlayer (generic function with 1 method)

- `skipPlayer(arr::Array) = duplicate(arr)[3:2+length(arr)]`

```
nextPlayer (generic function with 1 method)
```

```
• nextPlayer(arr::Array) = vcat(arr[2:end], arr[1])
```

```
build_dueling_network (generic function with 1 method)
```

```
• build_dueling_network(network::Chain) = begin
```

```
run_MultiAgent (generic function with 1 method)
```

```
• run_MultiAgent(policy::AbstractPolicy, env::AbstractEnv, stop_condition,
  hook::AbstractHook) = begin
```

Simple Blackjack

Note: In this simplified game, aces are always 1.

Rules

- Player gets 2 cards, dealer gets 1 (Player can see own 2 cards and dealer's first card)
- At each turn the player must decide whether to hit (draw another card) or stand (end turn)
- After each player turn, the dealer will draw a card
- If the player stands, the dealer will draw cards until their sum is at least 17
- The game ends when:
    - The dealer cannot draw any more cards
    - The dealer goes bust (total>21)
    - The player goes bust (total>21)
- If the dealer goes bust, player wins; if the player goes bust, dealer wins
- If neither player nor dealer goes bust, the highest point value wins
    - If player goes bust or loses, reward is -1
    - If player wins with <21, reward is 1
    - If player wins with 21, reward is 2
- Player can see total value of own hand and dealer's hand

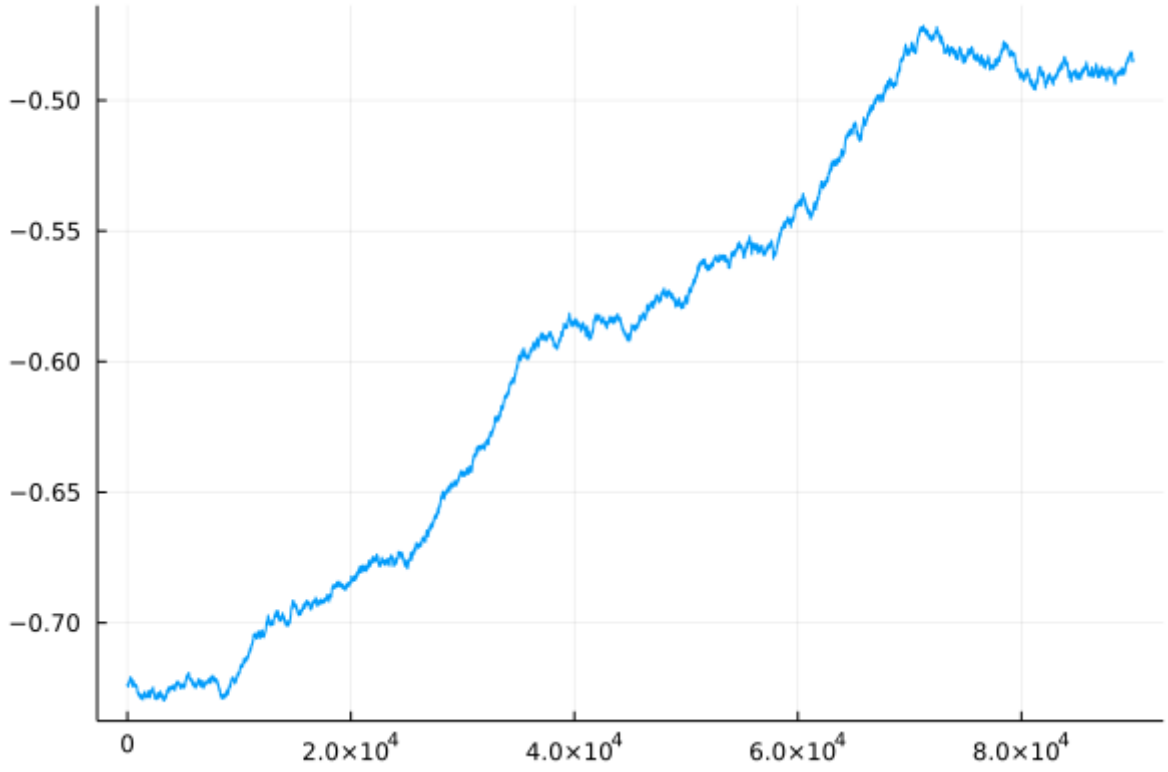
- begin

```
test_SimpleBlackjackEnv (generic function with 1 method)
```

```
test_SimpleBlackjackEnv() = begin
```

```
rewards_blackjack =
 [-1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0,
```

```
rewards_blackjack = test_SimpleBlackjackEnv()
```




```
plot(moving_average(rewards_blackjack, convert(Int, N_STEPS/10)), legend=false)
```

Single-Player ERS

The purpose of this environment is to test how quickly agents can learn rules

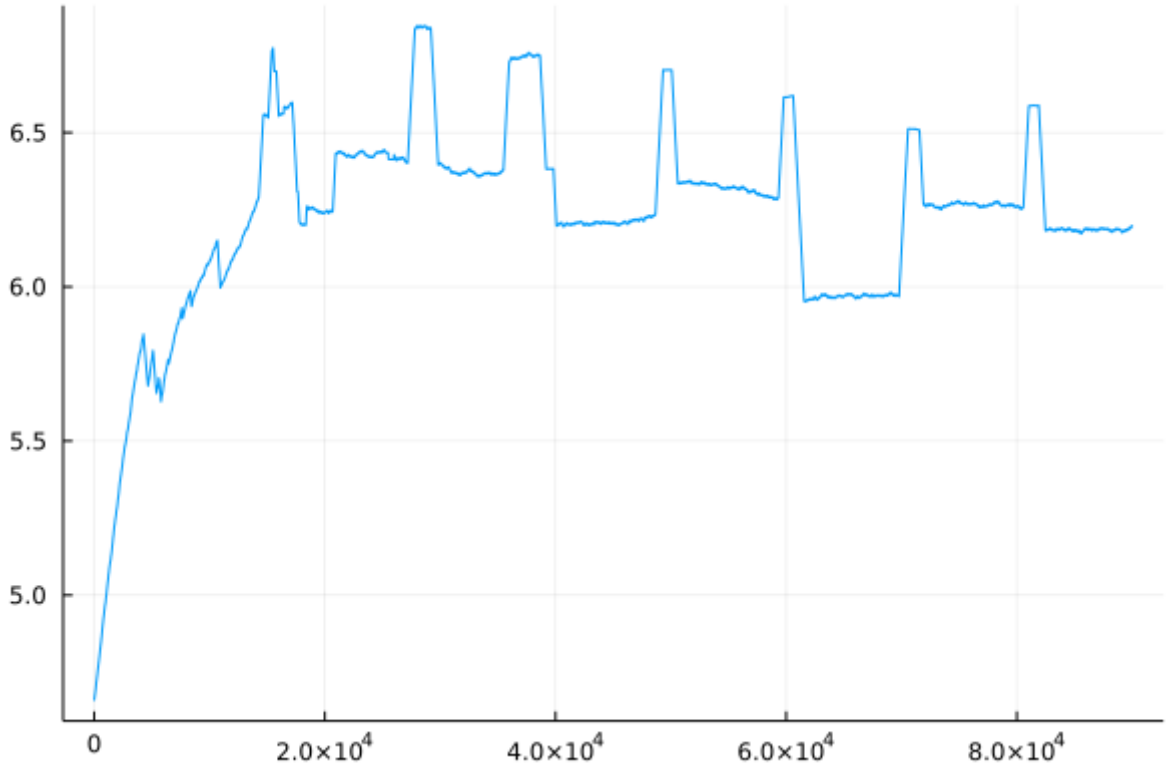
Rules

- Player will place one card face up
- n other "players" will continue to place cards face up
    - Players is in quotes because they will only play cards, not try to win the game
- Player can choose to hit/wait after each turn
    - If a player hits correctly, they will receive all of the cards in the pile; a hit is valid if:
        - Top two cards are the same value
        - Top card and 3rd card are the same value (X-Y-X...)
        - Top card and bottom card are the same value
        - Top two cards are king and queen (order does not matter)
        - Top two cards add to 10
    - If a player hits incorrectly, they will put one card from their hand into the pile
- If a player places a face card (A, K, Q, J), the next player must play either 4(A), 3(K), 2(Q), 1(J) cards. If the next player does not play a face card before the limit, the previous player gets the pile.
- Player wins the game by accumulating all of the cards
- Reward will be determined by the number of cards the player has after 10 turns

- begin

```
test_SinglePlayerERSEnv (generic function with 1 method)
```

```
test_SinglePlayerERSEnv(n::Int) = begin
```




```
begin
```

Modified Spoons

Rules

- This is a single-player modification of the game Spoons focused only on the card-passing aspect of the game
- Player will have 4 cards
- At each turn, player will get a card and then pass one card to the bottom of the drawpile
- The game will end when the player creates a "book" (all 4 cards in the hand have the same value)
- Reward will be determined by how many cards are the same

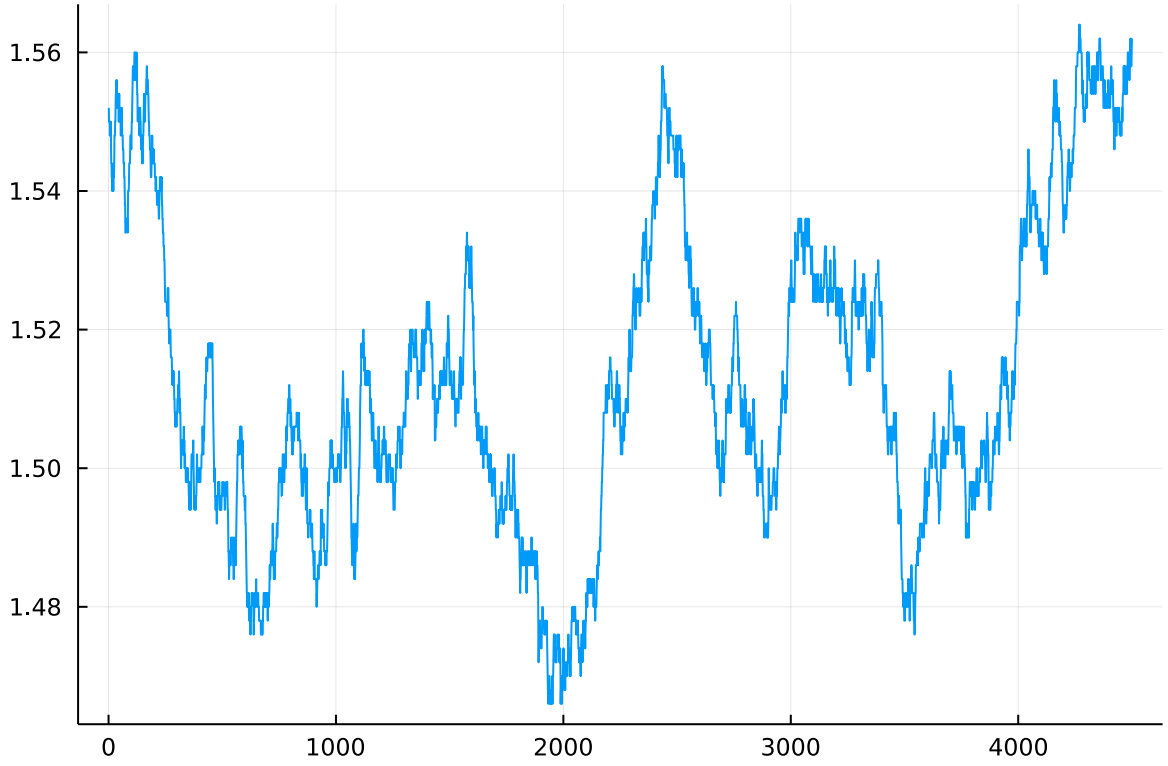
- begin

```
test_ModifiedSpoonsEnv (generic function with 1 method)
```

```
test_ModifiedSpoonsEnv() = begin
```

```
rewards_SPOONS =
 [2.0, 1.0, 1.0, 2.0, 2.0, 1.0, 2.0, 1.0, 2.0, 3.0, 2.0, 1.0, 2.0, 2.0, 2.0, 2.0, 1.0, 2.0, 1
```

```
rewards_SPOONS = test_ModifiedSpoonsEnv()
```




```
plot(moving_average(rewards_SPOONS, convert(Int, length(rewards_SPOONS)/10)), legend=false)
```

Strategic War

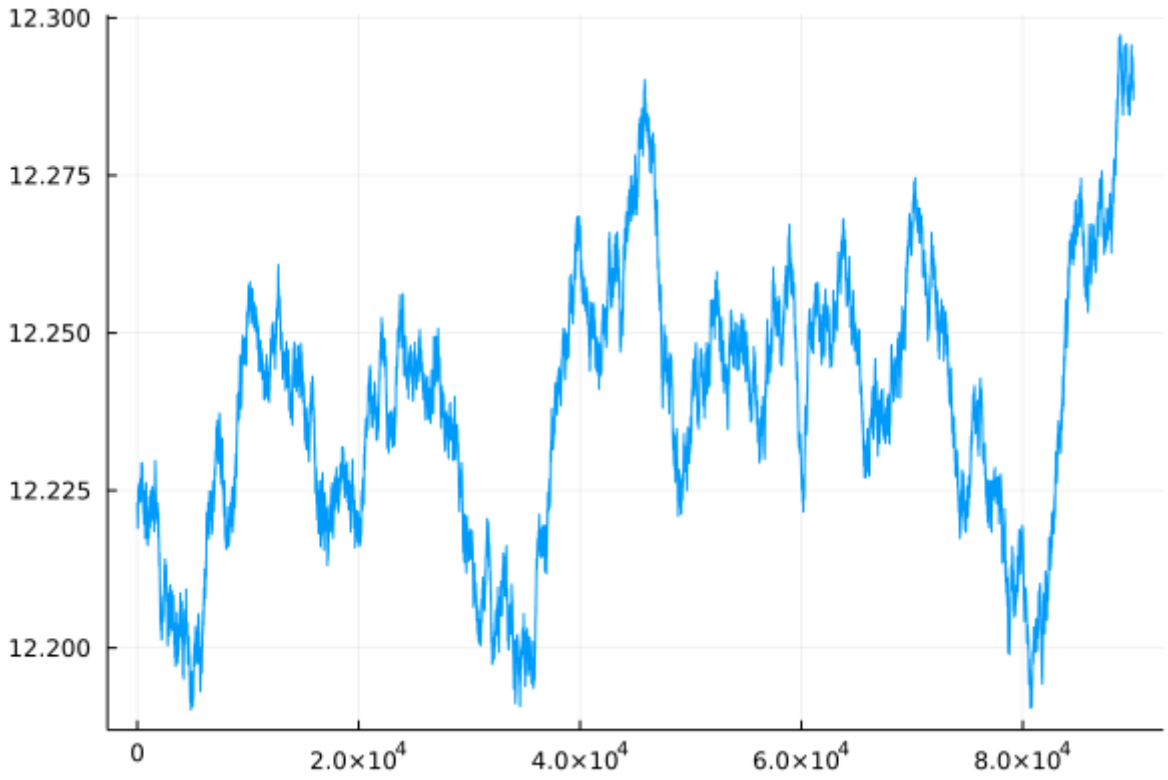
Rules

- Player1 and Player2 receive 26 cards each
- At each turn:
  - Player1 and Player2 will play a card simultaneously
  - The player who placed the higher value card will receive a point
  - The played cards will be discarded
- The reward of a player is how many points they have when all 52 cards have been played
- State will be represented by an array of the number of cards of each number a player has

- begin

```
test_StrategicWarEnv (generic function with 1 method)
```

```
test_StrategicWarEnv() = begin
```




```
begin
```

UNO

Rules

- Each player receives 7 cards to start
- At each turn, if a player has a playable card (same color/value of top card or wild), that player must play it
- If a player does not have a playable card, they must draw a card
    - If the drawn card is playable, they must play it
- Some cards have powerups (for the sake of simplicity, wild will always switch to the color that the player who played it has the most cards of)
- The first player to run out of cards wins

- begin

```
test_UnoEnv (generic function with 1 method)
```

```
test_UnoEnv(n) = begin
```

```
    )
```

```
DefaultTestSet("random policy with UnoEnv", [], 2000, false, false)
```

```
RLBase.test_runnable!(UnoEnv(3)) # if this runs, the environment works
```

Libraries Used (References)

```
@article{bezanson2017julia,
  title        = {Julia: A fresh approach to numerical computing},
  author       = {Bezanson, Jeff and Edelman, Alan and Karpinski, Stefan and Shah, V
iral B},
  journal      = {SIAM review},
  volume       = {59},
  number       = {1},
  pages        = {65--98},
  year         = {2017},
  publisher    = {SIAM},
  url          = {https://doi.org/10.1137/141000671}
}
@misc{Tian2020Reinforcement,
  author       = {Jun Tian and other contributors},
  title        = {ReinforcementLearning.jl: A Reinforcement Learning Package for the
Julia Programming Language},
  year         = 2020,
  url          = {https://github.com/JuliaReinforcementLearning/ReinforcementLearnin
g.jl}
}

@article{Flux.jl-2018,
  author    = {Michael Innes and
               Elliot Saba and
               Keno Fischer and
               Dhairya Gandhi and
               Marco Concetto Rudilosso and
               Neethu Mariya Joy and
               Tejan Karmali and
               Avik Pal and
               Viral Shah},
  title     = {Fashionable Modelling with Flux},
  journal   = {CoRR},
  volume    = {abs/1811.01457},
  year      = {2018},
  url       = {https://arxiv.org/abs/1811.01457},
  archivePrefix = {arXiv},
  eprint    = {1811.01457},
  timestamp = {Thu, 22 Nov 2018 17:58:30 +0100},
  biburl    = {https://dblp.org/rec/bib/journals/corr/abs-1811-01457},
  bibsource = {dblp computer science bibliography, https://dblp.org}
}

@article{innes:2018,
  author    = {Mike Innes},
  title     = {Flux: Elegant Machine Learning with Julia},
  journal   = {Journal of Open Source Software},
  year      = {2018},
  doi       = {10.21105/joss.00602},
}

@misc{
  author    = {Invenia Technical Computing},
  title     = {Intervals},
  year      = {2020},
  url       = {https://github.com/invenia/Intervals.jl}
}

@misc{
  author    = {"Rafael Fourquet <fourquet.rafael@gmail.com>"},
  title     = {StableRNGs},
  year      = {2020},
  url       = {https://github.com/JuliaRandom/StableRNGs.jl},
}
```

```
@misc{
  author    = {"Tom Breloff (@tbreloff)"},
  title     = {Plots},
  year      = {2021},
  url       = {https://github.com/JuliaPlots/Plots.jl},
}
```