Technical Report/ AI Learning Manual

DLG_Summer 2019

Team Typhoon

Contributors:

David Llanio

Hirad Peyvandi

Gautam Banuru

Pranav Banuru

Ashwin Char

Ishika Majumder

Table of Contents

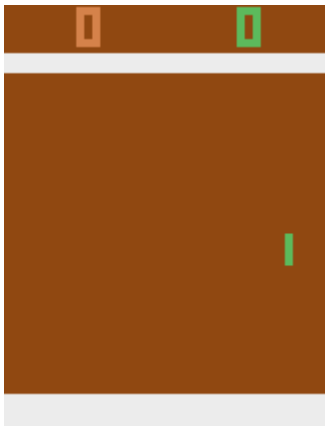
Introduction

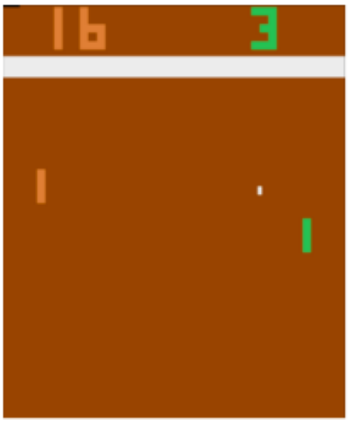
SmartPong is a program created by Andrej Karpathy. This program implements artificial intelligence to make it play itself and win. SmartPong outputs lines of code to see the efficiency of the AI as the number of episodes, or games, increase.

Through this project, we learn the foundations of Artificial Intelligence by analyzing this operated program. In this project, we analyzed the Atari game called Pong, and through the reverse engineering technique, we learned how we can train an AI model to master the game pong through the deep neural network using reinforcement learning algorithm.

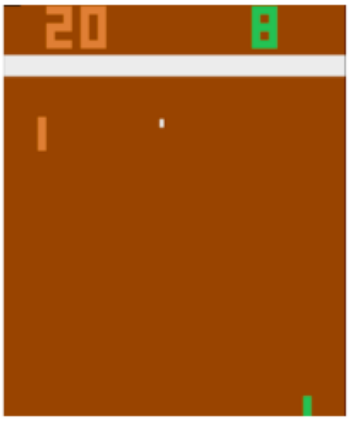
Figuring out how to run the program and what software and libraries were required to overcome the errors was one of the early challenges we faced. This program and many other AI programs are written in python programming language due to its simplicity. Therefore, the first application we installed was the latest version of python which is python 3.7. When you open the program, you see all these errors at the very beginning of the code where it's asking you to install the numpy, gym, and gym(Atari) library.

After we installed these libraries separately, in order to bring them all into one place and make use of them in our program, we installed Anaconda software which simplifies package management and enables the user to create their own environment and use a single environment to run the program. By this time, we must have had everything ready to run the program, but when we tried to do so, we were still getting an error from the gym library. After some research, we realized that the Gym library requires visual C++ Build tool, so we also had to install the visual studio for C++ to get that gym library running. Now we had all the necessary software, libraries, and packages installed and all we had to do to run the program and get some actual results was to make a few changes to the code because the code was written in Python version 2.7 and we were using python version 3.7.

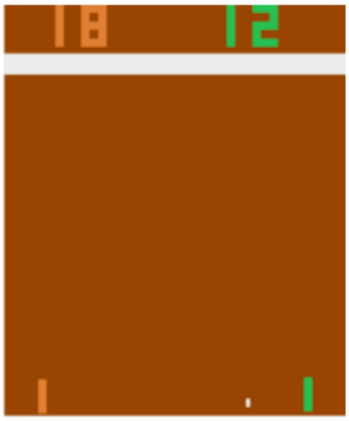




Ep: 15



Ep: 391



Ep:635

Abstract

This Technical Document serves as a guide for individuals to learn more about Artificial Intelligence, specifically Karpathy's SmartPong. It includes a high-level description of SmartPong for advanced students and professors. We also show how to install and run Karpathy's Smartpong code.

The version of the code being used in the analysis is the modified version. The code is broken down into mini sections and thoroughly analyzed with key concepts that are commonly used AI. We placed detailed descriptions of these key concepts prior to the analysis as a reference for readers. Finally, the conclusion proves an efficient AI is achievable and an overview of this project and our work.

Installation and Running

Video Guide: https://www.youtube.com/playlist?list=PLwvt34ierZxIesNGya03gq5Xh48IQfj0m

Windows

1. Install Python 3.7 from https://www.python.org/, make sure to add it to PATH.
2. Install Anaconda from https://www.anaconda.com/, register Anaconda as the system Python 3.7
3. Download the Smartpong code from https://gist.github.com/karpathy/a4166c7fe253700972fcbc77e4ea32c5 as a ZIP file and extract the python script.
4. Download and install Microsoft Visual Studio from https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=Community&rel=16&rid=30015
5. Download C++ Build tools for Visual Studio from https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=BuildTools&rel=16, video recommended for this part.
6. Download and Install Git from https://git-scm.com/download/win
7. Look for the Anaconda prompt application in your windows search bar.
8. Once Anaconda prompt is open type: “pip3 install gym numpy” excluding the quotation marks, press enter. This will install the gym and numpy libraries.
9. Then type: “pip3 install git+https://github.com/Kojoley/atari-py.git” excluding the quotation marks, press enter. This will install the gym[atari] library.
10. Find the Smartpong code in your download folder, should have the “pg.pong.py” name, right-click it and select “Edit with NotePad++”.
11. Make the following changes:
    Line 3: import _pickle as pickle
    Line 24: grad_buffer = { k : np.zeros_like(v) for k,v in model.items() } # update buffers that add up gradients over a batch
    Line 25: rmsprop_cache = { k : np.zeros_like(v) for k,v in model.items() } # rmsprop memory
    Line 43: for t in reversed(range(0, r.size)):
    Line 117: for k,v in model.items():
    Line 125: print ('resetting env. episode reward total was %f. running mean: %f' % (reward_sum, running_reward))
    Line 132: print ('ep %d: game finished, reward: %f' % (episode_number, reward), '' if reward == -1 else ' !!!!!!!!')
12. Save the file.
13. Right-click the file and select “Edit with IDLE”
14. Press run on the top bar, and then press “run module”. The program should run.
15. Press control+c to cancel program run.

MacOS

1. Download and install Python 3.7 from https://www.python.org/
2. Download and install Anaconda from https://www.anaconda.com/
3. Download the Smartpong code from https://gist.github.com/karpathy/a4166c7fe253700972fcbc77e4ea32c5
4. Open Anaconda and go to the environment tab, then press the create button on the bottom and create a new environment, make sure python 3.7 is selected as the package.
5. With the new environment selected go to home and press the install button for VS code, Visual studio code.
6. When the installation of VS code is complete, launch the application from Anaconda.
7. Press the file tab at the top and open the Smartpong code in visual studio.
8. Make the following changes:
   Line 3: import _pickle as pickle
   Line 24: grad_buffer = { k : np.zeros_like(v) for k,v in model.items() } # update buffers that add up gradients over a batch
   Line 25: rmsprop_cache = { k : np.zeros_like(v) for k,v in model.items() } # rmsprop memory
   Line 43: for t in reversed(range(0, r.size)):
   Line 117: for k,v in model.items():
   Line 125: print ('resetting env. episode reward total was %f. running mean: %f' % (reward_sum, running_reward))
   Line 132: print ('ep %d: game finished, reward: %f' % (episode_number, reward), '' if reward == -1 else ' !!!!!!!!')
9. Under open editors right-click on the file and select “open in terminal”
10. Type: “pip install numpy” excluding the quotation marks to install the numpy library, press enter.
11. Type: “pip install gym” excluding the quotation marks to install the gym library, press enter.
12. Type: “pip install gym[atari]” excluding the quotation marks to install numpy press enter.
13. Close the terminal and save the file.
14. Repeat step 9.
15. Type: “python3 X.py” excluding the quotation marks where X is the name of the file, press enter. The program should run.
16. Press control+c to cancel the program run.

Terms and Concepts

Machine Learning

Machine learning is divided into three categories. Supervised learning, Unsupervised Learning, and Reinforcement Learning. In the supervised learning algorithm, we feed the data as the input to our model as well as the expected output, then we let our model come up with the fastest and the most efficient algorithm to get the desired output. In the unsupervised learning algorithm, we feed the data to our model as the input and let the algorithm to find a relationship between data and categorizes them.

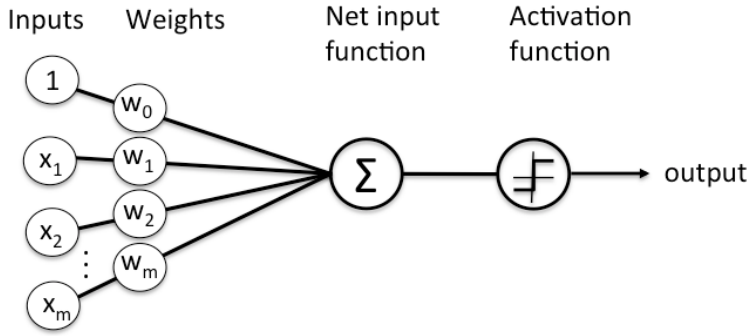
In reinforcement learning, we feed the data to our model as the input and compare the accuracy through objective function then we change the model's parameters and repeat the operation. In other words, in reinforcement learning, we have an agent which learns how to behave in an environment by performing actions from the results. For instance, in the smartPong program, we get a snapshot of the game pong and give this data as the input to our model, our model decides whether to go up or down and finally we check whether our model has won the game to get a reward of +1 or lost the game to give a -1 reward through objective function and based on those rewards, our model modifies its behavior through optimization algorithm.

Neural Networks

Neural networks contain algorithms that are modeled after the human brain in order to recognize patterns. They classify ungrouped data by finding similarities to the sample set of data. Humans group examples of data in order to be used as the "foundation" for the neural network to find patterns. Any labels that humans can identify can be used to train neural networks. Although classification requires labels, deep learning does not.

Deep learning can also be considered as unsupervised learning. Unsupervised learning models have more chances of being accurately trained because these often train on large sets of data which help increase precision.

The networks are composed of nodes that are in each layer. Nodes assign numerical significance to input data. They also combine the input with coefficients, also known as weights. These products are then summed and passed through an activation function that determines whether or not the product will progress further.



In deep neural networks, each layer of nodes trains a certain set of features based on the previous layer. This results in precision. The deeper into the neural net that you go, the more advanced elements you will see because each layer is building off the previous. When training, nodes try to reconstruct the input from samples over repetitions in order to minimize error. Neural networks find similarities and what the represent, whether it is based on labeled data or is a complete reconstruction.

When starting to train, neural networks will create models based on a collection of weights (coefficients). This tries to model data's relationship to labels in order to grasp the data's structure. When learning, the Input is out into the network and the weights connect that input with a set of guesses. The network then takes the guess and compares it to its true classification to assume error. The weights are then adjusted to decrease error.

$$input * weight = guess \mid ground\_truth - guess = error \mid error * weight's$$

$$contribution\ to\ error = adjustment$$

These are the three key functions of neural networks: scoring input, calculating error, and updating the model. A neural network also uses linear regression.

$$y = bx + a$$

With many input variables, the formula looks like this:

$$y = b_1 * x_1 + b_2 * x_2 + b_3 * x_3 + a$$

This multiple linear regression will happen at every node of the neural network. The inputs are combined with different coefficients in different proportions.

Policy Network

The network that transforms input frames to output actions. The simplest method to train a policy network is with a policy gradient.

Policy Gradient (Descent)

Gradient Descent is also known as the optimization function. Gradient is also known as "slope" and represents the relationship between the 2 variables. Because the neural network is trained to test out different weights, the relation between the *error* and the *weights* is the derivative *dE/dw.* This measures the slightest degree change that affects error. These are passed through an activation function, sees if there is a change in error, and checks how weights affect activation. This formula sums up the relationships.

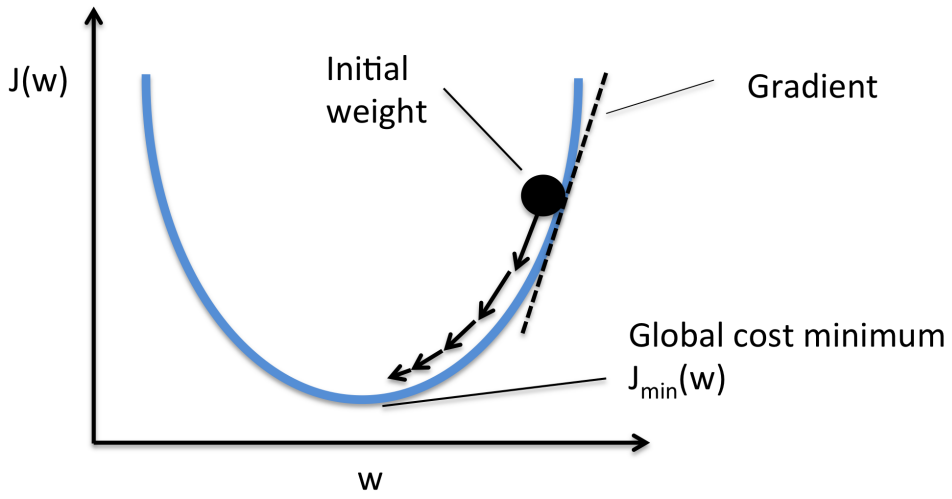
$$\frac{dError}{dWeight} = \frac{dError}{dActivation} * \frac{dActivation}{dWeight}$$

There are numerous parameters that keep updating in order to reach peak optimization. Gradient descent is typically used when parameters cannot be calculated analytically, with algebra, but rather when an algorithm is necessary. The coefficients are exchanged in order to find the lowest and best cost.

The initial values will start at 0.0 or some random small value. The derivative is calculated in order to find the rate of change of the function. Executing the derivative makes it

easier to find the coefficient values by helping limit the domain. Next, α is used to define the learning rate. This specifies how much the coefficients can change on each update. If the change is too drastic, accuracy may be lost.

$$J(coefficient) = cost \quad | \Delta = \frac{d}{dw} cost | \quad coefficient = coefficient - (\alpha\Delta)$$



These steps are repeated until the cost gets as close to 0 as possible. It is recommended to plot cost vs time. By plotting the cost values per iteration, you will be able to clearly see if the gradient descent run is decreasing. If it is not decreasing, the learning rate should be reduced.

Rewards

The Reward Hypothesis is considered the "maximization of the expected value of the cumulative sum of a received scalar signal". The objective is to maximize this set of rewards. In order to do this, one must look into the Markov Decision Process (MDP).

In simple terms, the MDP states the probability of transitioning into a different state while getting some reward depending on the current state. In other words, any action in the future is only dependent on the present, not the past. Adding these rewards of different levels of

significance with rewards from the future result in discounted returns. A higher discount factor leads to higher sensitivity for rewards.

Sparse and Dense Rewards

For sparse rewards instead of receiving a reward at the end of every step/action, we receive a reward for the end of an episode. The agent learns what part of the episode action sequence led to the reward in order to imitate that sequence in the future. Sparse rewards are sample inefficient meaning the reinforcement learning program needs lots of training time before executing useful behavior.

For a dense reward system rewards are given at the end of every step of an episode. The program is influenced by immediate rewards rather than working towards a long term goal implying a shallow lookahead. SmartPong will be using a sparse reward system.

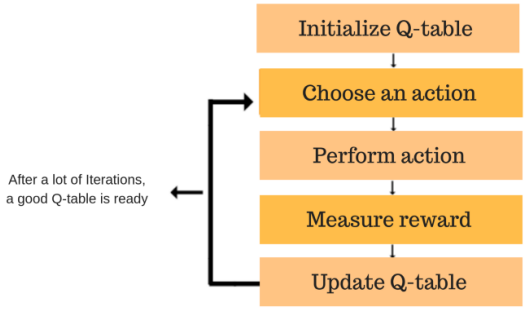
Discount Factor

Determines the importance of the accumulated future events in the MDP model. A higher γ leads to a higher sensitivity for rewards in future steps. To prioritize rewards in the distant future, keep gamma closer to one. A number closer to one will consider the previous results; however a number closer to zero will not learn anything. A discount factor closer to zero indicates that only rewards in the immediate future are being considered, implying a shallow lookahead. A factor equal to or greater than 1 will cause the convergence of the algorithm. However high enough discount factors (ie. $\gamma = 0.99$) result in similar intertemporal preferences with the same behavior as having a discount factor of 1. The optimal discount factor largely depends on whether a sparse reward ($\gamma = 0.99$ most optimal) or dense reward ($\gamma = 0.8$ may perform better) system is in place. If one wants to weigh earlier experience less they may try a myopic function where gamma grows linearly from 0.1 to a final gamma (ie. $\gamma = 0.99$ or 0.8) as a function of the total timesteps.

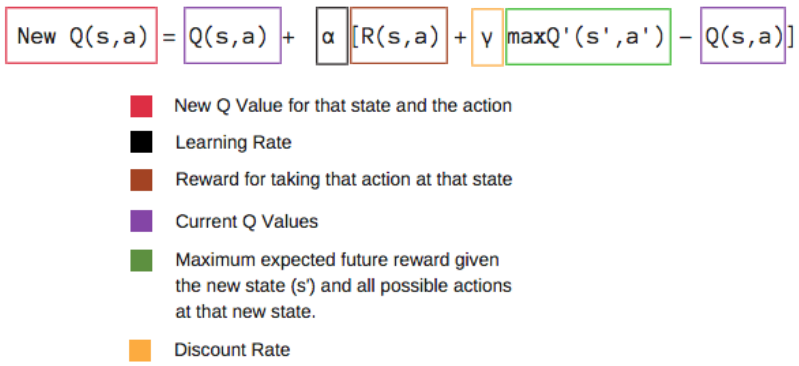
Q-Learning

Q-Learning is considered an off-policy algorithm because it learns from actions that occur outside the current policy. Its goal is to learn a policy that maximizes reward. The Q stands for quality which represents the usefulness of a certain action in gaining a reward.

First, you must create a matrix containing [state, action] called a q-table. The values must be initialized to 0. After each episode, these values will be updated. The agent can either Explore or Exploit. Exploiting includes using the q-table as a reference to select actions based on the maximum value. Exploring entails selecting actions at random and discovers new states. When updating, 3 steps take place: the agent starts in a state and takes action to receive an award, the agent selects an action based on the max value or at random, then updates the q-values.



These values are adjusted based on the difference between discounted values and old values. The new values are discounted using a discount factor, also known as γ.



Hyperparameters

Hyperparameters are constant values that must be set before the learning process begins. These values can be tuned to influence our model's learning performance.

A hyperparameter with too high a value will struggle to decrease the validation error. A hyperparameter with too low a value will take extremely long to learn. The most efficient values are often found through trial and error, with a human inputting test values rather than automatic adjustments. Tuning/optimizing the hyperparameter values properly to the task needed will lead to more accurate learning for a model.

Hyperparameters are optimized to excel at a specific task (ex. training an AI to play Pong) so they will differ among different tasks (ex. training an AI to recognize dogs and cats).

Input Layer

The input layer is the first layer of an artificial neural network. Each node in the input layer passes its value to each node of the first hidden layer and is multiplied by weights from the hidden layer nodes. The sum of these weights is passed through the activation function of each individual hidden layer node.

Backpropagation

An algorithm that uses the error of an instance to adjust the weights of nodes to decrease error in future instances. (an instance refers to one run of the code) Propagation in an AI situation represents sending/transmitting information. When an AI makes a prediction, it will have some errors. Backpropagation takes this error and uses it to send **back** changes to the weights, in an effort to decrease the error. Backpropagation uses **Gradient Descent** to determine whether to shift the weight values up or down.

Batch Size

The number of episodes the AI will gather experience for before updating its weights through **backpropagation** for the next batch.

With a batch size of 10, the AI will run 10 episodes and use the data collected from those 10 episodes to improve its accuracy during the next batch run. This repeats.

Learning Rate

Learning rate is a hyper-parameter that controls how much weights will change during **backpropagation**. **Gradient Descent** will help to determine the best direction of weight value change (up or down), but the learning rate helps determine how much to shift.

The smaller the Learning Rate, the longer computation time is and vice versa, but too large a learning rate (in the context of the model) can lead to decreased learning accuracy, despite the shorter computation time.

Learning rate is referred to as α.

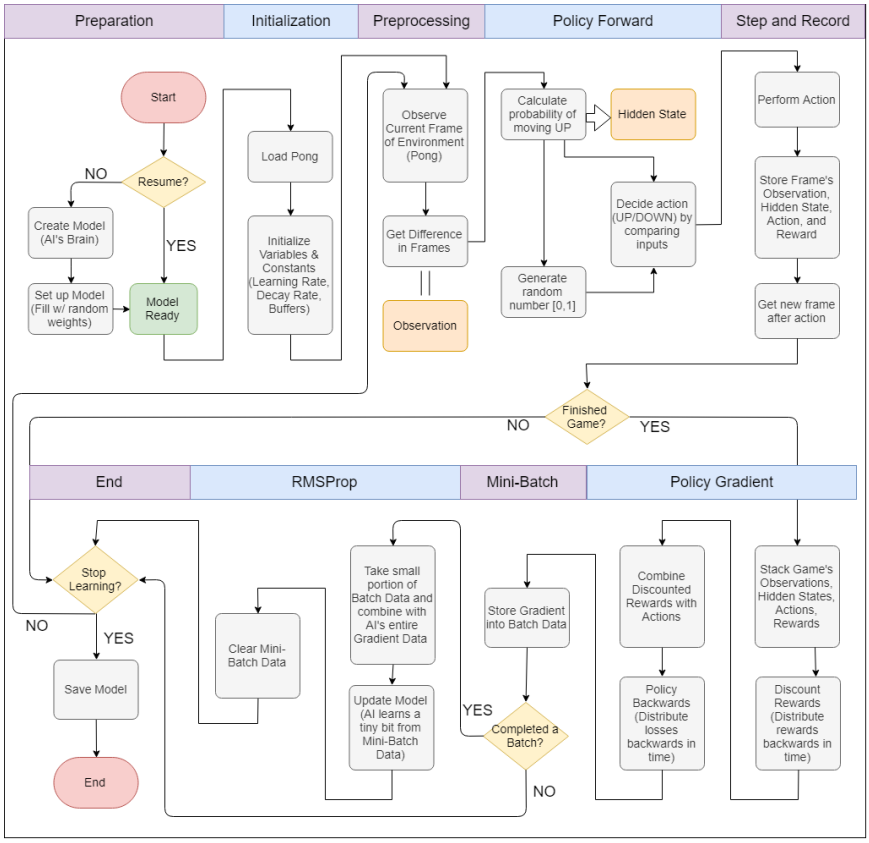
Xavier Initialization

Xavier Initialization assigns random weight values to all neurons in the model. Rather than having completely random weights to begin, Xavier initialization sets weights to be assigned randomly based on a hyperbolic tangent function (ie. the standard normal distribution curve) so that the variance of the weights is 1. This will guarantee that the weights will not begin as too large or too small. If this happens, then the neurons will become saturated, and they will be rendered nearly useless due to their dynamic range and inadequate representational power. All weights chosen through Xavier Initialization lay between 0 and 1.

ReLU Nonlinearity

ReLU stands for rectified linear unit. ReLU is a common activation function where ReLU is linear for all positive values and zero for all negative values. Mathematically it is defined as y = max(0, x). It's sparsely activated meaning since ReLU is zero for all negative inputs, it likely won't activate the neuron at all. This means neurons can be dedicated toward the specifics in an image such as identifying the ball vs. paddles. The downside is the "dying ReLU" problem where a ReLU neuron is "dead" if it's stuck in the negative side and always outputs 0. Once a neuron becomes zero, it's unlikely to recover, leaving large parts of the network unused. A solution is the "leaky ReLU" which has a small slope for negative values.

Analysis of SmartPong Code



Overview of the Smartpong code.

```
import numpy as np
import _pickle as pickle
import gym
```

These are lines 2-4 (David) (GB checked)(AC checked)

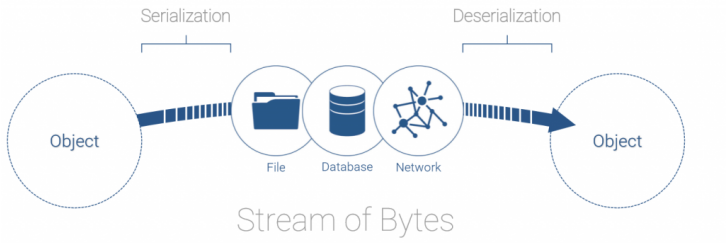
Here we are importing the NumPy, _pickle, and gym libraries. So here is an overview of what they do:

NumPy:

Within this library comes a lot of mathematical tools that the Smartpong code uses to create the artificial intelligence. The ones specifically used throughout the code are “random” which is a function that returns a random number, “sqrt” which is used to get the square root, “zeroes_like” which is a function that returns an array of zeros, “exp” which is used to calculate the exponential of all elements of the array inputted into it, “float” which is the data type that allows decimal numbers, “dot” which gets the dot product of two arrays, “outer” which calculates the outer product of two vectors, “zeroes” which returns an array of zeros with a selected shape and type, “vstack” which stacks 1 dimensional arrays (Also Known As vectors) vertically by row, “mean” which returns the average of the input data, and “std” which computes the standard deviation.

Pickle:

Within this library are tools that implement binary protocols for serializing and de-serializing a Python object structure. The term pickling refers to the process of a python object hierarchy converting into a byte stream, unpickling is the inverse. Modules used from the _pickle library throughout the program include: “load” which reads a pickled object representation from the open file object file and returns the reconstituted object hierarchy specified therein, and “dump” which writes a pickled representation of obj to the open file object file.



Gym:

This library was created by the AI research company OpenAI. Within are tools to create and structure reinforcement learning algorithms. One of the tools used from this library is the "make" module which imports a pre-made simulation of the pong game for the Smartpong AI to use as its learning environment. The Pong simulation takes actions as input and returns the next frame of the game. This saves the user from creating complicated environments to experiment with reinforcement learning.

```
# hyperparameters
H = 200  # number of hidden layer neurons
batch_size = 10  # every how many episodes to do a param update?
learning_rate = 1e-4
gamma = 0.99  # discount factor for reward
decay_rate = 0.99  # decay factor for RMSProp leaky sum of grad^2
resume = False  # resume from previous checkpoint?
render = True
```

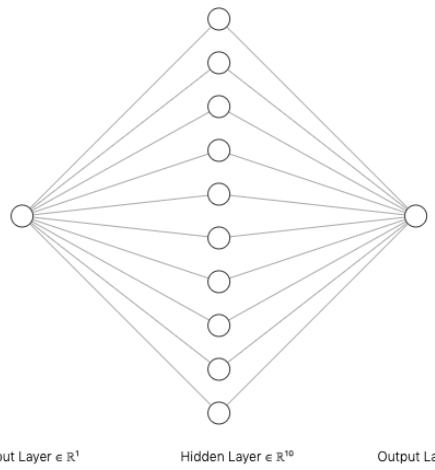
These are lines 6-13 (Gautam)(DL checked)(AC checked)

Hyperparameters are constant values set before the learning process begins. These values can be tuned to vary the results of our model's learning performance. While some hyperparameters are best under certain thresholds (like gamma) which have been determined through decades of research, in many cases getting the right hyperparameters for a specific model is found through trial and error. Each learning model has a specific goal to achieve. Hyperparameters are optimized to excel at a specific task (ex. training an AI to play Pong) so they will differ among different tasks (ex. training an AI to recognize dogs and cats).

A neural network has an input layer, a number of hidden layers, and an output layer. There will be x number of nodes in each layer. Our model will have 1 node in the input layer, 1 hidden layer with 200 nodes and 1 node in the output layer. 200 nodes in the hidden layer were chosen selectively because it had good performance for the task at hand. Generally, an extreme amount of hidden layer nodes will cause overfitting, while too few nodes will give the model a hard time learning. We want the neural network to learn patterns rather than memorize all possible moves. Notice we have one input layer and one output layer. The input layer will take a frame from the pong game (an 80x80 image consisting of 6400 pixels) and the output layer will return the probability of moving the AI paddle up (a decimal between 0 and 1).

A complete Pong game is called an episode. At the end of the episode, we will receive a reward of +1 (our AI scored) or -1 (opponent scored). Otherwise, during each step, or frame, of the Pong game we will receive a reward of 0. We record the rewards as a history of the game performance. For our AI to perform better we must update our model through backpropagation

using the reward history to guide what actions will be taken in the future. Backpropagation is a computationally expensive task. As such we will perform backpropagation every 10 episodes hence a batch size of 10.



Neural Network with 1 Hidden Layer (http://alexlenail.me/NN-SVG/index.html)

The learning rate determines how quickly our model will learn. If our model learns too fast it may overfit and not recognize patterns as well. It is generally kept at 1x10^4.

A higher gamma (γ), or discount factor, leads to a higher sensitivity for rewards in future steps. To prioritize rewards in the distant future we keep gamma closer to one. A number closer to one will consider the previous results; however a number closer to zero will not learn anything. A discount factor closer to zero indicates that only rewards in the immediate future are being considered, implying a shallow lookahead. The optimal discount factor largely depends on whether a sparse reward (γ = 0.99 most optimal) or dense reward (γ = 0.8 may perform better) system is in place. We are acting on sparse rewards with our model because we only receive rewards at the end of the episode (hence no immediate feedback) so we will keep gamma at 0.99.

The decay rate determines the ratio between old experience kept and new experience gained when updating the weights of the model via the RMSProp algorithm. In our case decay_rate = 0.99 so each time the model is updated 99% it is devoted to keeping old experience

and 1% is devoted to new experiences. This is further explained on lines 115-121 where the decay rate is applied.

The Resume flag determines whether to load a previous model saved via pickle or start a new model from scratch. Remember that backpropagation updates our model with the experience it has gained. We don't need to train from scratch every time (that would be silly). Since this is the first time we are using the program we will set Resume to False.

The Render flag determines whether to display the visuals of the SmartPong program. Since we want to see the Pong game in action in real-time via the Gym environment we will set Render to True.

```
# model initialization
D = 80 * 80  # input dimensionality: 80x80 grid
if resume:
   model = pickle.load(open('save.p', 'rb'))
else:
   model = {}
   model['W1'] = np.random.randn(H, D) / np.sqrt(D)  # "Xavier" initialization
   model['W2'] = np.random.randn(H) / np.sqrt(H)
```

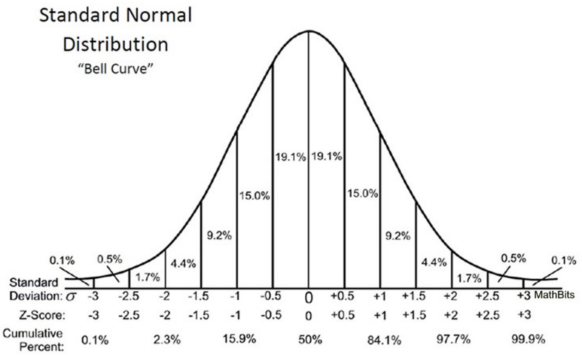
These are lines 15-22 (Ashwin) (GB checked)(DL checked)

These lines are used to initialize and assign values to the model which will be used throughout the SmartPong code. This model will contain weights that will be used for forward propagation, and will be updated through backward propagation, or backpropagation. In order to create a working and efficient model with weights that will allow for the correct decision to be made in a short training period, the model must be initialized in the most optimized way possible.

First, the dimensionality of the image that will be used must be specified. The image that will be used in SmartPong will be 80x80 grid or 6400 pixels. It is important to define these dimensions due to how the model is initialized, which will require the size of the image.

SmartPong allows for the agent to resume training from a certain checkpoint. This is where the resume boolean, initialized in the previous section, will be used. If the “resume” variable is set to true, then the model will be initialized from a previous model. This means that the weights that were created and updated in previous trainings will be loaded and used for the current training. This way the model can continue training off of the past experience it has gathered. This is made possible by the use of the “pickle” library and more specifically the “load” module. The “load” module will read in a pickled file and convert it into an object hierarchy. Within the “load” module exists an “open” module which allows for which will open the file named “save.p” in “rb”, which means read in binary.

If the “resume” variable is set to false, which it is in this case, then the model will be initialized using a method known as Xavier initialization. Rather than having completely random weights to begin, Xavier initialization allows for weights to be assigned randomly based on a hyperbolic tangent function (ie. the standard normal distribution curve) so that the variance of the weights is 1. This will guarantee that the weights will not begin as too large or too small. If this happens then the neurons will become saturated, and they will be rendered nearly useless due to their dynamic range and inadequate representational power.



The model will be a dictionary. Dictionaries are used to store keys and their corresponding values which will be useful in keeping track of each layer of the model. The “W1” key is the first one initialized and will be the weights connected to the first hidden layer. It is initialized by choosing random weights from a standard normal distribution curve in the shape 200x6400. Each weight is then divided by the square root of the grid size. This ensures that the variance will be 1, the essential part of Xavier initialization. This layer will be used to detect game scenarios.

The weights “W2” connected to the second layer are then initialized using random number on a standard normal distribution curve with a shape of 200x1. These weights are divided by the square root of the number of neurons. These result in a variance of 1. This second layer will be responsible for helping determine whether to move up or down.

```
# update buffers that add up gradients over a batch
grad_buffer = {k: np.zeros_like(v) for k, v in model.items()}
rmsprop_cache = {k: np.zeros_like(v)
                 for k, v in model.items()}  # rmsprop memory
```

These are lines 24-25 (David) (GB checked)(AC checked)
Here we are creating the gradient buffer variable and the rmsprop_cache variable.

The gradient buffer variable, also known as the replay buffer, is used by the AI to store tuples of the experience that AI gets through every episode of learning. The experience tuples that are stored are the state, the action, the reward, and the next_state.

Because the Smartpong program is using policy gradients, the different data we get in the different experience tuples are not independent of each other, meaning that 1 data set from each tuple must be together in batches for the AI to learn properly. The AI then takes batches from the buffer variable, or replay buffer, to train the AI.

Then the rmsprop cache is created which will be keeping track of the per-parameter sum of squared gradients. This sum comprises of previous adjustments done to the weights of the neural network, and because of the structure of the on.

RMSprop Formula:

The RMSProp algorithm is considered “leaky" because it “leaks" the previous estimates to the current calculation with the decay_rate set at 0.99. This means 99% is the old value stored in rmsprop_cache and 1% is the new value received from the square of the gradient. This ensures the model builds off of previous experience when updating weights.

$$E[g^2]_t := 0.99\, E[g^2]_{t-1} + 0.01\, g^2_t$$

```
def sigmoid(x):
   # sigmoid "squashing" function to interval [0,1]
   return 1.0 / (1.0 + np.exp(-x))
```

These are lines 27-28 (Ashwin) (GB checked)(DL checked)

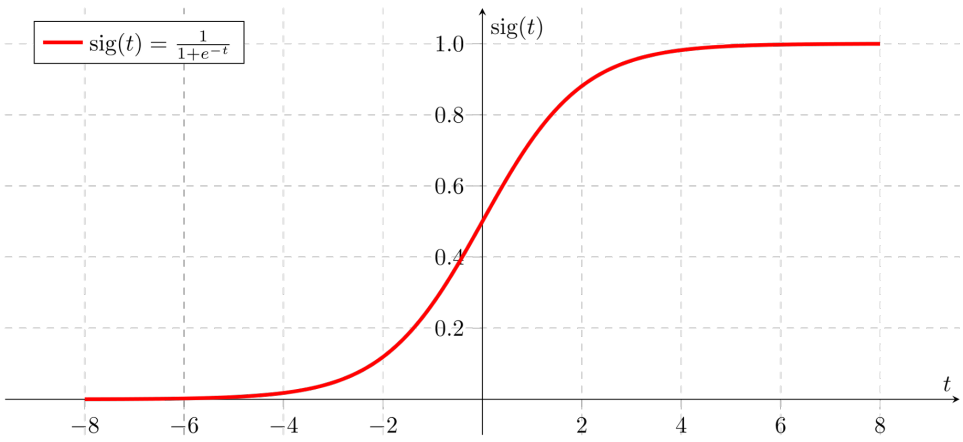
These lines define the sigmoid function, which is a function that is used to "squash" any inputs into an interval of (0,1). The function is simply $f(x) = 1/(1 + e^{-x})$. SmartPong uses this function in order to generate log probabilities of moving up from values given by the neural network. This occurs in the "policy_forward" function, which also utilizes the ReLU nonlinearity to return the hidden states and probabilities.

Sigmoid benefits:

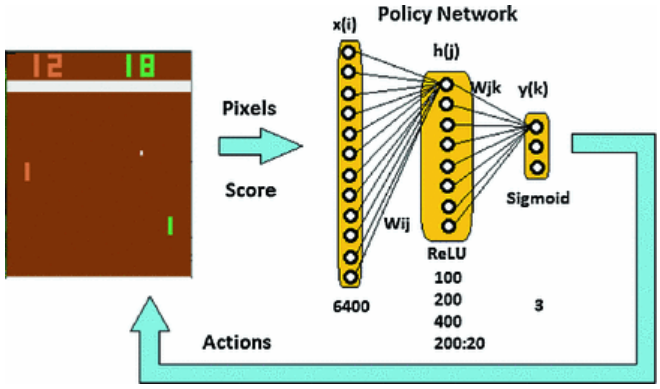
SmartPong uses the sigmoid function due to its ability to always return numbers between 0 and 1, no matter its input. Along with this, the general S shape of this function creates a general steepness that occurs in the middle of the function which allows most inputs to tend to lean towards either 0 or 1, making it binary. The sigmoid function is a nonlinearity, making it a very useful and popular activation function. Compared to the step function and general linear function, sigmoid will always have a set range of (0,1) which are the 2 choices the agent has, moving UP or Down. Sigmoid is also monotonic, which activation function must be.

Potential shortfalls:

Unfortunately, sigmoid is not a perfect activation function. One potential shortfall of using sigmoid is the possibility of a vanishing gradient. As inputs become infinitely large or small, different inputs will result in largely the same output. Although this may not be terribly destructive to SmartPong, neural networks with a larger number of layers can experience these smaller gradients, which make training an agent extremely inefficient.



The graph shown is the sigmoid function, which is one of the activation functions used in SmartPong.



Pixels and scores are taken as input and sent through the policy network, the network then outputs a probability that is determined using sigmoid and is used to come up with an action in the game.

```
def prepro(I):
    """ prepro 210x160x3 uint8 frame into 6400 (80x80) 1D float vector """
    I = I[35:195]  # crop
    I = I[::2, ::2, 0]  # downsample by factor of 2
    I[I == 144] = 0  # erase background (background type 1)
    I[I == 109] = 0  # erase background (background type 2)
    I[I != 0] = 1  # everything else (paddles, ball) just set to 1
    return I.astype(np.float).ravel()
```

These are lines 30-37 (Gautam)(DL checked)(AC checked)

Preprocessing reduces the complexity of an image down to its necessary components. The components we want are only those that affect the actions taken by the Pong AI. That means we can remove the scoreboard, any unused space, and color from the image. Reducing the image complexity will also help our neural network train faster because there is less (unnecessary) information to process.

An image in Python is stored as an array of arrays. Each inner array represents a row and the i-th pixel in each row corresponds to the i-th column in the image. We can select a specific pixel with the syntax Image[row, column, channels]. This is known as index notation.

The image “I” starts off with the shape 210x160x3 uint8. This means the image has 210 rows, 160 columns, and a depth of 3 representing the color channels in the order Blue, Green, Red. Notice this is different from the traditional RGB pattern due to historical reasons. Each element in the image is a uint8 data type which means it is an integer from 0-255 (meaning there are 256 or 2^8 possible values).

First, we crop the image to include only rows 35-194. This removes the scoreboard and the border separating the scoreboard from the playing arena (rows 0-34) as well as the bottom border (rows 195-210).

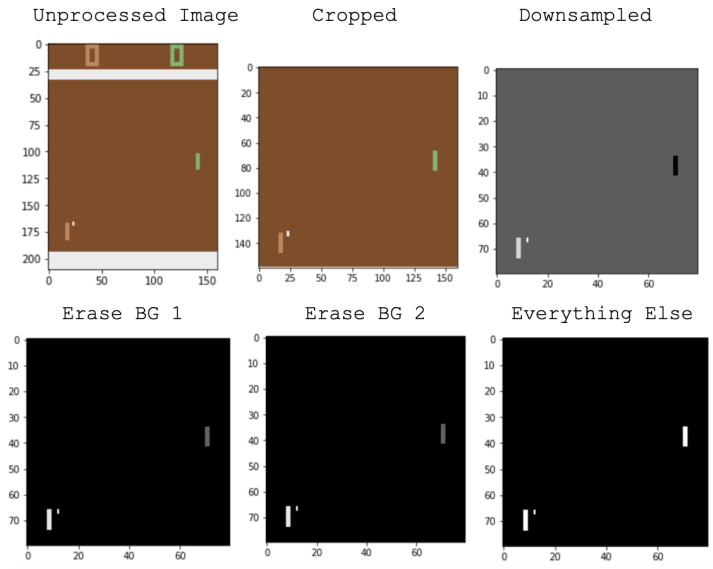
Downsampling reduces the complexity of the image. We do this by removing every other row and column from the image. Let’s look at the first value in I[::2, ::2, 0] which is ::2. Indexing

uses the syntax start:stop(inclusive):step. In ::2 there is no “start” and “stop” so no rows are cropped and we skip every second row (a “step” of 2) starting from the first row. The “0” means we only include the blue channel (depth of 0) in this downsample. This updates the image size to 80x80. The blue channel values effectively act as a grayscale value.

Our goal is to have a binary image to further reduce complexity. The most important part of the image is the position of the ball and paddles. Everything else can be considered noise to be filtered out. The next two actions remove the background colors. I[I == 144] = 0 checks every value in the image and those equal to 144 are set to 0. Likewise for I[I == 109] = 0.

The last action I[I != 0] = 1 will set every value in the image not equal to 0 to be 1. This directly identifies the ball and paddles.

Finally, we turn the 2D image array into a 1D vector of floats (decimal values which in our case will be 0.0 or 1.0) with I.astype(np.float).ravel() because a vector is the preferred way our neural network processes information. Floats are used because these values will interact with the weights in our network which are also floats. This returns a single vector with 6400 values.



Visualizations of each step in prepro(I).

```
def discount_rewards(r):
    """ take 1D float array of rewards and compute discounted reward """
    discounted_r = np.zeros_like(r)
    running_add = 0
    for t in reversed(range(0, r.size)):
        if r[t] != 0:
            # reset the sum, since this was a game boundary (pong specific!)
            running_add = 0
        running_add = running_add * gamma + r[t]
        discounted_r[t] = running_add
    return discounted_r
```

These are lines 39-47 (David) (GB checked)(AC checked)
Here, the function to calculate the discount rewards of the current episode is created.

Every episode of learning that the AI is going through is a game of pong, wherein the two players win the episode by scoring 21 points on the other player. In between the time that a player is scoring a point on the other player, there are actions that lead to one player scoring a point on another player. The AI is giving a good reward for every action, hitting the ball with the paddle by going UP, DOWN, or NO MOVEMENT, that the AI leads to a score, and giving a bad reward for actions that don't lead to a score.

These rewards are then discounted with the discount factor (γ) which, in this case, is 0.99. This discount factor, in particular, makes the AI weigh the immediate rewards in the episode more favorably than later rewards.

Here is the Discount reward formula:

$$total\ discounted\ reward = \sum_{i=1}^{T} \gamma^{i-1} r_i$$

In the function definition r is a 1D array of arbitrary size where each element in r is the reward given to the agent at each step of the running episode. You can think of a step as a frame in the pong video game. The vast majority of steps will have a reward of zero. We will move

through this list r backward to recognize the precursor steps that led to a nonzero reward. We will apply a discounted reward to those precursor steps via list discounted_r that grows from left to right until it reaches a nonzero reward step.

In essence we will create a list of zeros the same shape as r, set a count we will use to hold discount values as discount_add, move through the range backwards ie. right to left, reset the count if we reach another nonzero reward, compute the discounted reward (gamma is a number between 0 and 1 called the discount factor. Note the discount reward will decrease on successive iterations of the for loop), save discounted reward for specific step, and discount_r will now be a list that recognizes steps that led to a nonzero reward in list r.

For example,
if r = [0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0] and
gamma = 0.99 then
discount_rewards(r) = [0.99, 1.0, 0.9703, 0.9801, 0.99, 1.0, 0.0, 0.0]

These discounted rewards are then normalized by subtracting the mean and dividing by the standard deviation before being fed through backpropagation to optimize the weights of the neural network.

```
def policy_forward(x):
    h = np.dot(model['W1'], x)
    h[h < 0] = 0  # ReLU nonlinearity
    logp = np.dot(model['W2'], h)
    p = sigmoid(logp)
    return p, h  # return probability of taking action 2, and hidden state
```

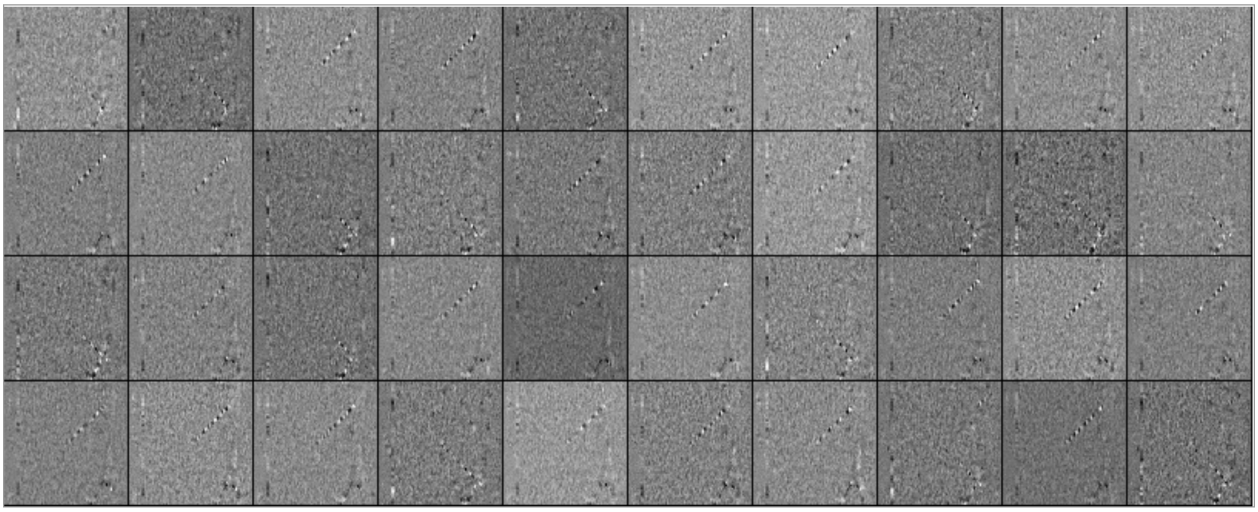
These are lines 49-54 (Gautam)(DL checked)(AC checked)

When we want to get a prediction from our neural network, we pass an input (ie. an image) and get an output (a probability). How a neural network makes a decision is ultimately a black box but we can discern the elements used to help the neural network come to a conclusion. policy_forward(x) describes our neural network model. We have an input layer with one node (x), a 1 hidden layer with 200 nodes, and an output layer with one node (logp). Each node in a layer is connected to every node in the next layer. Each connection stores a weight which is randomly initialized at the start between 0 and 1. These weight "layers" are comprised of model['W1'] and model ['W2']. Weights will be adjusted for better performance during backpropagation. Policy forwards are used to obtain predictions.

To get the initial output value at a node we multiply the weights connected to that node with all the outputs of the previous layer and sum those values together. This summed value is passed through an activation function to get the final output value for a node. The process repeats until we reach the output layer of a neural network. The purpose of an activation function is to only let values that meet specific criteria pass through the network. It imitates the fact that not all neurons in our brain activate for any given task.

Our input image x is an 80x80 2D numpy array flattened to shape (6400,1) from preprocessing. To get the output value at the hidden layer we take the dot product between the first layer weights of shape (200,6400) and image x. For 2-D vectors, it is the equivalent to matrix multiplication. A matrix multiplication would sum the weight multiplied by the input for all neuron connections from the last layer creating vector h. This returns a vector of shape (200,1) because for matrix multiplication shape AxB * shape BxC = shape AxC. Similarly, shape 200x6400 * shape 6400x1 = shape 200x1.We then pass all values through the ReLU nonlinearity

activation function to get the final out value for each node in the hidden layer. Any summed value less than zero becomes zero. Mathematically this is y = max(0, x). Notice that the first layer weights are actually of shape (6400,1). Backpropagation will update these weights to identify balls and paddles as patterns of 80x80 images like the ones shown below.



model['W1'] weights visualized

Now we take the output values from the hidden layer and repeat the process. Take the dot product of h and the second hidden layer weights. We pass these summed values through the sigmoid activation function acting on the output layer to produce the final probability of moving up as variable p. We then return p and h (which is the output values from the hidden layer) called the hidden state. We will use this hidden state later for backpropagation.

```
def policy_backward(eph, epdlogp):
   """ backward pass. (eph is array of intermediate hidden states) """
   dW2 = np.dot(eph.T, epdlogp).ravel()
   dh = np.outer(epdlogp, model['W2'])
   dh[eph <= 0] = 0  # backpro prelu
   dW1 = np.dot(dh.T, epx)
   return {'W1': dW1, 'W2': dW2}
```

These are lines 56-62 (Ashwin) (GB checked)

These lines make up the “policy_backward” function. While “policy_forward” dealt with propagation, this function is used in SmartPong as backwards propagation. The purpose of backpropagation is to compute the error for each layer of the neural network and then use that to update the individual weights. This function can be thought of as the actual learning that the machine is accomplishing. Although the propagation function produces probabilities that determine action, the back prop will “fix” the weights that are being used to determine the probabilities. The longer the agent is trained, the more that the weights are being adjusted and optimized for the best performance. Eventually, the model will be updated in lines 115-121.

SmartPong is able to accomplish this by using partial derivatives of our error function with respect to each weight. This begins in line 58 with “dW2” being set as the updated derivative with respect to the second weight matrix, which is responsible for determining the action. This is done using a dot product between the hidden layer’s transpose, columns become rows and vice versa, and the gradient with advantage. The final product is then flattened into a 1-D array using the “ravel” module.

Next “dh”, the derivative of the hidden state, is given the value of the cross product between the gradient with advantage and the second weight matrix in the model. The ReLU activation function is then applied to “dh” meaning if the hidden state is negative, it is given the value of 0. The ReLU activation function is also used in “policy_forward”, so to maintain consistency, it used again in the back prop.

“dW1”, the derivative of the first weight matrix, is given the value of the dot product of the derivatives of the hidden state’s transpose and “epx” which is defined in line 99, and stands for the input observations. The first weight matrix is responsible for recognizing game scenarios, hence the use of observations.

Now that all the partial derivatives have been calculated, the weights must be updated accordingly. Line 62 accomplishes this by creating a dictionary and mapping the W1 and “dW1” values together, as well as “W2” and “dW2”. This will effectively update the weights so that they are progressively becoming more effective at generating probabilities and their corresponding actions.

```
env = gym.make("Pong-v0")
observation = env.reset()
prev_x = None  # used in computing the difference frame
xs, hs, dlogps, drs = [], [], [], []
running_reward = None
reward_sum = 0
episode_number = 0
while True:
   if render:
       env.render()
```

These are lines 64-72 (David) (GB checked)(AC checked)

Here the environment where the AI will be learning is created. The environment is the pong game simulation imported from the gym library using the "make" module. The specific simulation of Pong that is going to be used is one that is structured to give images, screenshots, or frames of the game as the input into the neural network. Then the observation variable is the environment-specific object representing your observation of the environment. The observation is set to the initial form of the environment at the end of every episode of the game to prepare for the next episode.

The environment is created as a class. The current state (xs), hidden state (hs), action gradient (dlogps), and discount reward sum of episodes (drs) are the things that will be recorded inside of the environment and will be used to make the AI learn to play pong. The current state will consist of the frame of the game where an action is going to be taken by the AI, and where other observations will be recorded. The hidden state of the game will consist of the calculations of the policy gradient and the weight parameters of the neural network. The action gradient will be the log probability of the AI taking the UP action, which is to move the paddle on its side of the Pong game UP. The discount reward sum of the episode will be the sum of all the discounted rewards in each of the episodes.

The running_reward, as the name implies, will be running over all the games and episodes that take place and will be recording the average sum of rewards in every game and episode, which helps gauge if the AI is performing well or not. If there is a positive change in

this running_reward then the AI is learning and playing better. The reward sum will be the sum of the rewards in each episode (not discounted rewards).

Then, the learning will begin. First, a counter is made for the episode number and an infinitely running while loop is created which will run the learning code until the user chooses to cancel the program. The render part is if the render variable is set to true in the hyperparameters section of the program, the program will display the actual game of Pong and the AI learning inside of it on a separate window.

```
# preprocess the observation, set input to network to be difference image
   cur_x = prepro(observation)
   x = cur_x - prev_x if prev_x is not None else np.zeros(D)
   prev_x = cur_x
```

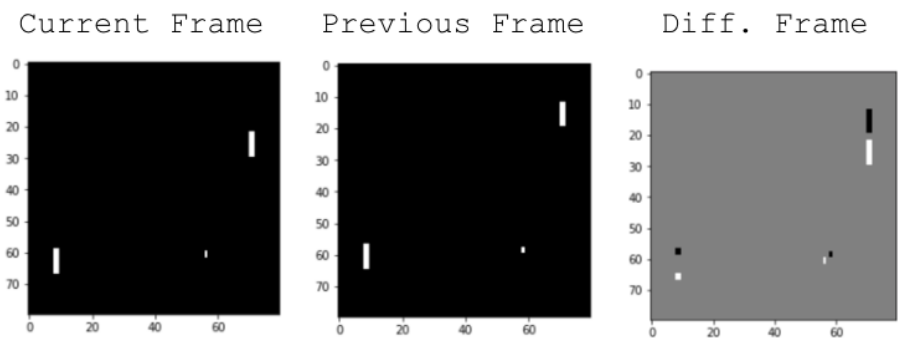
These are lines 74-77 (Ashwin) (GB checked)(DL checked)

The objective of these lines is to preprocess the observation so that it is a usable vector in the “policy_forward” function. This is accomplished by employing the “prepro” function, which takes in a 210x160x3 uint8 frame and returns a 1D float vector. Preprocessing the image only gives provides an image of the current frame and this isn’t going to be useful in determining game scenarios and probabilities. In order to make our preprocessed image useful, a difference in images is necessary.

To gain a difference image, which will be given the variable “x”, the previous frame, “prev_x”, must be utilized along with the “cur_x” variable that is given the value of the current preprocessed frame described earlier. Finding the difference frame is as simple as subtracting the previous frame from the current frame; however, there is one specific instance where this will not work. “prev_x” was given the initial value of None, meaning there is nothing to subtract. This indicates that the first frame of the first episode will not have a difference image.

To avoid this, an if statement is created to see if the value of “prev_x” is not equal to None. In this case, it will simply subtract the 2 images. In the case where “prev_x” does equal None, then the difference variable ”x” is set equal to an 80x80 vector of zeros.

In order to have a previous frame for the next iteration, we must assign the previous frame variable the value of the current frame. This is exactly what line 77 is accomplishing. It sets “prev_x” equal to “cur_x”. There is no need to replace “cur_x” with anything because it will be given a different preprocessed observation for the next iteration through the while loop.


Current Frame
Previous Frame
Diff. Frame

```
# forward the policy network and sample an action from the returned
probability
   aprob, h = policy_forward(x)
   action = 2 if np.random.uniform() < aprob else 3  # roll the dice!
```

These are lines 79-81 (Gautam)(DL checked)

We want our Pong AI to make the right decisions on whether to move the paddle up or down. This action is solely determined based on the weights in the neural network. When we do a policy forward we are asking the neural network what to do based on the input we give it. We are not training the network (ie. updating the weights via backpropagation). The neural network in policy_forward() will give us the probability of moving the paddle up (aprob). Remember that our neural network has two hidden layers. The neural network will take an input (in our case the difference image x), compute the sum of the weights multiplied by the input for each node, pass the value through the activation function, and repeat the process for each of the hidden layers. The hidden state (h) is the dot product of the input and the first hidden layer weights to be used later in policy_backward() (ie. backpropagation) for, you guessed it, updating the weights.

Now that we have aprob (the probability of moving the paddle up), we must actually decide if we should move the paddle up. This action will be given to the Pong simulation in Gym via env.step(action) which changes the state of the environment and produces the next frame of the game after the action is taken. The possible actions are to move the paddle up (action = 2) or move the paddle down (action = 3). We decide the action with np.random.uniform() which produces a number between 0 and 1. We set action = 2 if aprob is greater than this number and otherwise action = 3. The larger aprob is, the more likely action = 2 will be chosen. The reason we use probabilities to determine the action rather than discrete binary values from the neural network is to provide variability so that the AI can “explore” the environment by giving a chance for action = 3 even when aprob is high. Exploring allows the AI to discover new movement patterns that could lead to a higher score and for such to be reflected when the weights are updated in backpropagation.

```
# record various intermediates (needed later for backprop)
   xs.append(x)  # observation
   hs.append(h)  # hidden state
   y = 1 if action == 2 else 0  # a "fake label"
   # grad that encourages the action that was taken to be taken (see
http://cs231n.github.io/neural-networks-2/#losses if confused)
   dlogps.append(y - aprob)
```

These are lines 83-87 (David)(GB checked)

Here the current state of the game or the current frame is recorded into the xs list which is for current states and the hidden state h of the current frame is recorded in list h. The lists xs and hs will be used in backpropagation which occurs in the policy_backward() function.

So, to understand the fake label mentioned in the Smartpong code it has to be understood what an action label is in supervised learning. An action label in supervised learning is an identified action the supervised AI can apply to new input data. For example, if a supervised AI is trained to spot a horse in a picture, it creates a horse label by being trained with multiple pictures of horses, and it looks for horses in new pictures based on the labeled data it compiled while being trained. Now, in the Smartpong program, and reinforcement problems in general, there is no labeled data that the AI can base its actions on. So, a fake label is made to determine the action with advantage (y) after receiving the action the AI will make from the previous section. In this case, y will be 1 if the action taken was 2 (going up) and will be 0 if the action taken was any other action.

Next, the 1 or 0 that was assigned to the y variable is subtracted by the action probability (aprob) variable value. This creates a new value which will be either positive or negative that will be used to update the weights of the neural network and is recorded in list dlogps. Updating the weights like this helps the AI converge faster, in other words, reach its peak performance.

```
# step the environment and get new measurements
   observation, reward, done, info = env.step(action)
   reward_sum += reward

   # record reward (has to be done after we call step() to get reward for
previous action)
   drs.append(reward)
```

These are lines 89-93 (Gautam)(DL checked)

Now that we know the action our AI should take we need to input this action into the Gym Pong environment. Remember action = 2 moves the paddle up and action = 3 moves the paddle down. env.step(action) inputs the action to be taken into the Gym Pong environment and env.step applies the action to create the next frame in the simulation as well as the other return values. We are given 4 return values from env.step(action).

observation (object): an environment-specific object representing your observation of the environment. (ie. the current frame of the pong game)

reward (float): amount of reward achieved by the previous action. The scale varies between environments, but the goal is always to increase your total reward. (ie. -1.0, 0, or 1.0)

done (boolean): whether it's time to reset the environment again. Most (but not all) tasks are divided up into well-defined episodes, and done being True indicates the episode has terminated. (ie. True or False)

info (dict): diagnostic information useful for debugging. It can sometimes be useful for learning (for example, it might contain the raw probabilities behind the environment's last state change). Official evaluations of your agent are not allowed to use this for learning.

Finally, the reward is added to the reward sum. We also append the reward into drs because we will later use drs to determine the string of events that lead to a +1 or -1 reward.

Observation Action Cycle:

This process highlights the relationship between our system and the environment. The system is given an observation from the environment. Using that observation, the system is able to generate some type of action. In the case of SmartPong, the observation is fed into the "policy_forward" function and produces a probability that will determine an action. This action will then be executed in the environment(moving UP or DOWN). Following this action, the environment produces and returns another observation, which continues this cycle of observations and actions.

```
if done:  # an episode finished
        episode_number += 1

        # stack together all inputs, hidden states, action gradients, and
rewards for this episode
        epx = np.vstack(xs)
        eph = np.vstack(hs)
        epdlogp = np.vstack(dlogps)
        epr = np.vstack(drs)
        xs, hs, dlogps, drs = [], [], [], []  # reset array memory
```

These are lines 95-103 (Ashwin)(DL checked)

After an episode is completed, there are a certain set of operations that must be carried out. These lines are concerned with recognizing when an episode has been completed as well as some of the initial operations allow for the data received from each step to be utilized effectively.

The duration of an episode of the SmartPong is defined to be how long it takes for our agent or the opponent to achieve 21 points. We can tell if an episode has been completed based on if the "done" variable, which assigned a value through the "step" module, is set to true or false. In order to execute the functions that are necessary after an episode has finished, SmartPong uses an if statement to evaluate if "done" variable is true.

It is also important to keep track of the episode number so that the progress of the agent can be interpreted and the learning rate can be better visualized. This is why immediately after each episode has been completed, the "episode_number" variable is incremented by 1. SmartPong prints out the reward sum and running reward after each episode, and seeing the episode number along with these quantities provides a greater understanding of how the agent is performing in comparison to other agents in SmartPong.

Following this, the data that was accumulated in the lists xs, hs, dlogps, and drs must be stacked so that they can be used in the backwards propagation as well as for calculating the discounted rewards. These lists contain accumulated data on the observation, the hidden states,

the log probability of moving up or down, and the rewards respectively. Stacking these lists will essentially convert them into vertical arrays. This is done so that the matrix functions in can be carried out properly. Operations can't be applied to an entire list; however, they can be applied to each cell of an array. The new arrays that will store the data are epx, eph, epdlogp, and epr. Due to the dot and outer products required in the backwards pass, they arrays had to be vertical rather than horizontal.

Each of the lists that were used in accumulating data from the previous episode must be cleared in order to collect new data for the next episode. Because of this each of the lists are reset to their original empty state, and the data from the previous game is stored in the new arrays that were created. Note, it is not necessary to reset epx, eph, epdlogp, and epr because these variables are first defined within the while loop.

```
 # compute the discounted reward backwards through time
       discounted_epr = discount_rewards(epr)
       # standardize the rewards to be unit normal (helps control the gradient
estimator variance)
       discounted_epr -= np.mean(discounted_epr)
       discounted_epr /= np.std(discounted_epr)
```

These are lines 105-109 (David)(GB checked)

Here the discounted rewards are being calculated and standardized. The program is using the discount_rewards function with the episode rewards (epr) as the input to the function, or parameter. As mentioned in a previous segment, the discount rewards help the AI figure out which of it's actions during the multitude of episodes will lead it to a win by discounting the rewards received for each action by the discount factor (γ) of 0.99. The discount factor in Smartpong of 0.99 makes the AI favor actions it took early on in the episode over actions taken later in the episode.

After the discount rewards are calculated, they need to be standardized by subtracting the mean the discount rewards and dividing by the standard deviation of the discounted rewards. This standardizes the rewards to be unit normal (ie. mean is 0 and standard deviation is 1) and will control the variance of the distribution. This is done to give more impact to outliers within the discounted rewards, good and bad, and to eliminate actions that had no impact or gave no reward. Sometimes if data inputs are outside a usable range the output from the neural network can be completely wrong.

```
# modulate the gradient with advantage (PG magic happens right here.)
epdlogp *= discounted_epr
grad = policy_backward(eph, epdlogp)
for k in model:
    grad_buffer[k] += grad[k]  # accumulate grad over batch
```

These are lines 111-113 (Ashwin)(DL checked)(GB checked)

These lines are concerned with changing the gradient as well as using the "policy_backwards" function for backpropagation These lines also create a gradient buffer variable so that it can be used in updating weights within the model. The purpose of changing the gradient is to update the log probability so that the gradient can be calculated with updated and optimized parameters.

First, in line 111, the log probability "epdlogp" is multiplied with "discounted_epr". This is done to update the log probability so that it can effectively alter the gradient. The SmartPong code specifically states that it will "modulate the gradient with advantage" and that the "PG (policy gradient) magic happens right here".

With the updated "epdlogp" variable, the gradient should now be calculated. This is done by using the "policy_backward" function, which handles backpropagation. The parameters "eph" and "epdlogp" are passed to the function. This modifies the weights of the hidden neurons of the neural network for the next episode. The "grad" variable also now stores values of the partial derivatives calculated during the backpropagation, along with their corresponding layer.

Now that "grad" has values, the "grad_buffer" variable can be updated. This is done by using a for loop to iterate through the model. During each iteration, the "grad_buffer" variable is updated by adding the specific value corresponding with the iteration in "grad" to itself. This will end up accumulating "grad" over the entire batch. SmartPong will eventually use "grad_buffer" along with the "learning_rate" and "rmsprop_cache" will be used to update the model so that it will work more effectively.

```
# perform rmsprop parameter update every batch_size episodes
    if episode_number % batch_size == 0:
        for k, v in model.items():
            g = grad_buffer[k]  # gradient
            rmsprop_cache[k] = decay_rate * \
                rmsprop_cache[k] + (1 - decay_rate) * g**2
            model[k] += learning_rate * g / \
                (np.sqrt(rmsprop_cache[k]) + 1e-5)
            # reset batch gradient buffer
            grad_buffer[k] = np.zeros_like(v)
```

These are lines 115-121 (Gautam)(DL checked)

Our goal is to update our model weights after a set number of episodes to reduce computation cost. Our batch size is 10 so we will perform a neural network model update every 10 episodes.

Notice the code has a single for loop. In the first iteration of the for loop, k is equal to ‘W1’ and v is equal to the weights stored in ‘W1’. In the second iteration of the for loop, k is equal to ‘W2’ and v is equal to the weights stored in ‘W2’. This means all the actions in the for loop are performed for both W1 and W2 separately.

Every finished episode runs backpropagation which returns the gradient of change for each weight in W1 and W2. The individual weight gradients for W1 and W2 accumulate in the grad_buffer on lines 113 for every episode in a batch. The gradient signifies how to nudge the weights in order to achieve better model performance.

While we could directly apply the grad_buffer to the model weights we will first apply rmsprop to the grad_buffer. Why rmsprop? Gradients of very complex functions like neural networks have a tendency to either vanish or explode as values propagate through the function. Rmsprop deals with the problem by using a moving average of squared gradients to normalize the gradient itself. This balances the step size by decreasing the step for large gradient to avoid exploding and increasing the step for small gradient to avoid vanishing.

The RMSProp algorithm is considered “leaky" because it “leaks" the previous estimates to the current one with the decay_rate set at 0.99. This means 99% is the old value stored in rmsprop_cache and 1% is the new value received from the square of the gradient. This ensures the model builds off of previous experience when updating weights.

Finally, before the model weights are updated a learning_rate (which was set at 1e-4) is applied to the grad_buffer divided by the square root of the sum of the rmsprop_cache and 1e-5. “learning_rate” controls the rate of learning. If the model learns too slowly in the beginning, it may overfit rather than learn the basics of pong. While not implemented here a dynamic learning rate could be applied to increase/decrease the learning rate as a function of total timesteps. “grad_buffer” is finally reset at the end so that is can be empty for the next batch of episodes.

```
  # boring book-keeping
        running_reward = reward_sum if running_reward is None else
running_reward * \
            0.99 + reward_sum * 0.01
        print('resetting env. episode reward total was %f. running mean: %f' %
              (reward_sum, running_reward))
        if episode_number % 100 == 0:
            pickle.dump(model, open('save.p', 'wb'))
        reward_sum = 0
        observation = env.reset()  # reset env
        prev_x = None
```

These are lines 123-129 (David)(GB checked)

This section of code focuses on keeping records of changing variables in order to monitor the training of the AI over episodes.

First part of this "book-keeping" section is creating the variable for the running reward and calculating it from the reward sum. If the running reward variable hasn't been created yet, it is created and set equal to the rewards sum. If the running reward variable was already created before, it is set to 1% of the reward sum. The running reward helps to see if the AI is learning over the thousands of episodes of learning it goes through. As the training starts the running reward will be around -21 or -20 because the AI is getting negative rewards for losing games, but as the training progresses this number will slowly increase.

The next part of this section has the model being saved every 100 episodes. This is done so if the training, under some circumstances, has to be stopped, the user can continue the training at another time from where it left off. To continue the training all that has to be done is set the "resume" variable in the hyperparameters section of the program to TRUE.

Finally, The reward sum, environment, and the previous state variable (prev_x) are reset to get ready for the next episode of training.

```
if reward != 0:  # Pong has either +1 or -1 reward exactly when game ends.
      print('ep %d: game finished, reward: %f' %
            (episode_number, reward), '' if reward == -1 else ' !!!!!!!!')
```

These are lines 131-132 (Gautam)(DL checked)

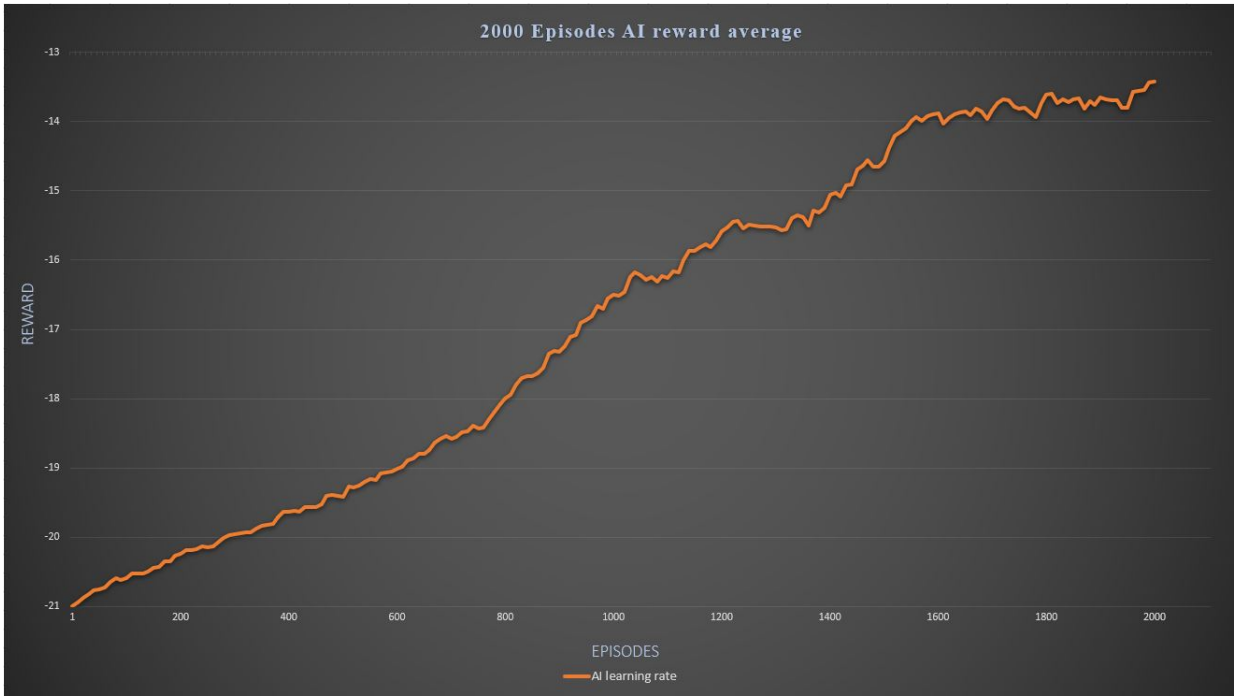
In order to keep the viewer updated on the AI's performance, we will provide updates in the form of print statements. Memorable points of interest are when a game has finished (which happens when reward = 1 or reward = -1). When reward = 0 the game is still in session. If reward = -1 this means the opponent scored against our AI so we will print the episode number and reward. On the other hand, if our AI scores a goal we will inform the user with "!!!!!!!!". Every time our AI scores our neural network is slowly becoming optimized and improving in performance.

Alternative Methods

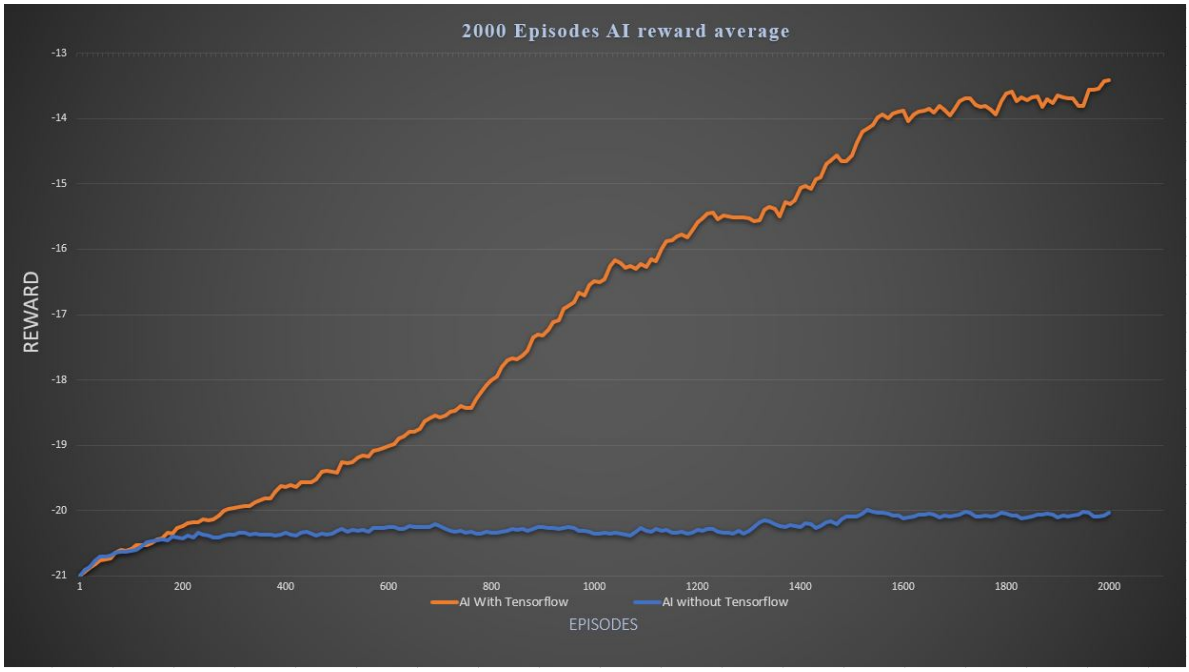
Tensorflow

Tensorflow is an open-source library that has been mostly used with Python. Now I have to say that TensorFlow is also available for Java or C or even Go language. Tensorflow library was designed by Google in 2015 and it helped the programmers to design an AI model a lot faster and easier through the very useful functions and variable structured that TensorFlow contains.

As an example, we see placeholder operation quite often in the code that we used TensorFlow components. Placeholder is simply a variable that we will assign data to at a later time. In other words, the Placeholder variable is used to create an empty space for our variable which currently we don't know what it is going to be, and it will be filled later in the code. For example, well first the left screenshot on this slide shows how we import the TensorFlow library as tf like we import other libraries. And the screenshot on the right-hand side shows how we declare input image as an x and output action as a y. when we declare these two variables we use the placeholder variable to define the type and the shape of our data as the parameters of the placeholder. The shape of the input image x will be represented by the 2d tensor of numbers with single-precision floating-point.



*This graph shows how well our model can learn the game pong through **TensorFlow** over **2000** episodes.*



*This graph shows how our AI model can learn the game pong with (orange) and without (blue) **TensorFlow** over **2000** episodes.*

Conclusion

Karpathy's "pong from pixels" code, or the Smartpong code, is a good example of a program that tackles a reinforcement learning problem, teaching an artificial intelligence to play the Pong game, by using policy gradients. While policy gradients are not the only solution to the reinforcement problem at hand, it certainly was an effective one. Through the dissection and analysis of the Smartpong code, one is able to have a valuable learning experience of how different concepts of artificial intelligence like learning rate, backpropagation, and policy gradients are put to practical use.

References

/@activatedgeek. “Policy Gradients in a Nutshell.” *Medium*, Towards Data Science, 2 June 2018, towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d.

/@aminamollaysa. “Policy Gradients and Log Derivative Trick.” *Medium*, Medium, 19 Nov. 2018, medium.com/@aminamollaysa/policy-gradients-and-log-derivative-trick-4aad962e43e0.

/@ardendertat. “Applied Deep Learning - Part 1: Artificial Neural Networks.” *Medium*, Towards Data Science, 9 Oct. 2017, towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6.

/@ashish_fagna. “Understanding OpenAI Gym.” *Medium*, Medium, 23 Mar. 2018, medium.com/@ashish_fagna/understanding-openai-gym-25c79c06eccb.

/@bushaev. “Understanding RMSprop - Faster Neural Network Learning.” *Medium*, Towards Data Science, 2 Sept. 2018, towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a.

/@gencozgur. “Notes on Artificial Intelligence, Machine Learning and Deep Learning for Curious People.” *Medium*, Towards Data Science, 5 Feb. 2019, towardsdatascience.com/notes-on-artificial-intelligence-ai-machine-learning-ml-and-deep-learning-dl-for-56e51a2071c2.

/@hafidz. “Understanding Learning Rates and How It Improves Performance in Deep Learning.” *Medium*, Towards Data Science, 27 Jan. 2018, towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10.

/@lskhere. “Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning.” *Medium*, Towards Data Science, 1 Aug. 2017, towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1.

/@m.alzantot. “Deep Reinforcement Learning Demysitifed (Episode 2) - Policy Iteration, Value Iteration and Q-Learning.” *Medium*, Medium, 8 Oct. 2018, medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-978f9e89ddaa.

/@m.alzantot. “Deep Reinforcement Learning Demysitifed (Episode 2) - Policy Iteration, Value Iteration and Q-Learning.” *Medium*, Medium, 8 Oct. 2018, medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-978f9e89ddaa.

/@sagarsharma4244. “Policy Networks vs Value Networks in Reinforcement Learning.” *Medium*, Towards Data Science, 7 Aug. 2018, towardsdatascience.com/policy-networks-vs-value-networks-in-reinforcement-learning-da2776056ad2.

“5. Data Structures¶.” *5. Data Structures - Python 3.7.4 Documentation*, docs.python.org/3/tutorial/datastructures.html.

"A Beginner's Guide to Backpropagation in Neural Networks." *Skymind*, skymind.ai/wiki/backpropagation.

"A Beginner's Guide to Deep Reinforcement Learning." *Skymind*, skymind.ai/wiki/deep-reinforcement-learning.

"A Beginner's Guide to Neural Networks and Deep Learning." *Skymind*, skymind.ai/wiki/neural-network.

FAtBalloonFAtBalloon 1, et al. "What Does the Hidden Layer in a Neural Network Compute?" *Cross Validated*, 1 Dec. 1963, stats.stackexchange.com/questions/63152/what-does-the-hidden-layer-in-a-neural-network-compute.

ihadannyihadanny 57511 gold badge66 silver badges1616 bronze badges, et al. "Why Do We Normalize the Discounted Rewards When Doing Policy Gradient Reinforcement Learning?" *Data Science Stack Exchange*, 1 Dec. 1967, datascience.stackexchange.com/questions/20098/why-do-we-normalize-the-discounted-rewards-when-doing-policy-gradient-reinforcem.

Karpathy, Andrej. "Deep Reinforcement Learning: Pong from Pixels." *Deep Reinforcement Learning: Pong from Pixels*, karpathy.github.io/2016/05/31/rl/.

McNulty, Eileen, et al. "What's The Difference Between Supervised and Unsupervised Learning?" *Dataconomy*, 10 June 2019, dataconomy.com/2015/01/whats-the-difference-between-supervised-and-unsupervised-learning/.

"Numpy." *PyPI*, pypi.org/project/numpy/.

"Numpy.dot¶." *Numpy.dot - NumPy v1.17 Manual*, docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html.

"Numpy.outer¶." *Numpy.outer - NumPy v1.17 Manual*, docs.scipy.org/doc/numpy/reference/generated/numpy.outer.html.

"Numpy.vstack¶." *Numpy.vstack - NumPy v1.17 Manual*, docs.scipy.org/doc/numpy/reference/generated/numpy.vstack.html.

OpenAI. "A Toolkit for Developing and Comparing Reinforcement Learning Algorithms." *Gym*, gym.openai.com/docs/.

Oppermann. "Self Learning AI-Agents IV: Stochastic Policy Gradient." *Medium*, Towards Data Science, 1 Aug. 2019, towardsdatascience.com/self-learning-ai-agents-iv-stochastic-policy-gradients-b53f088fce20.

"Pickle - Python Object Serialization¶." *Pickle - Python Object Serialization - Python 3.7.4 Documentation*, docs.python.org/3/library/pickle.html.

"Reinforcement Learning." *Exploring Science*, 7 June 2019, dashora7.wordpress.com/2019/06/07/reinforcement-learning/.

"Serialization Is Dead! Long Live Serialization!" *Waratek*, 6 Nov. 2018, www.waratek.com/serialization-is-dead-long-live-serialization/.

"Sigmoidal Nonlinearity." *DeepAI*, 17 May 2019, deepai.org/machine-learning-glossary-and-terms/sigmoidal-nonlinearity.

"The Markov Property, Chain, Reward Process and Decision Process." *Xavier Geerinck - Blog*, Xavier Geerinck - Blog, 20 May 2018, xaviergeerinck.com/markov-property-chain-reward-decision.

"Understanding Xavier Initialization In Deep Neural Networks." *PERPETUAL ENIGMA*, 7 Apr. 2016, prateekvjoshi.com/2016/03/29/understanding-xavier-initialization-in-deep-neural-networks/.

user2991243user2991243 1, et al. "What Is Batch Size in Neural Network?" *Cross Validated*, 1 Nov. 1965, stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network.

Woergoetter, Florentin, and Bernd Porr. "Reinforcement Learning." *Scholarpedia*, www.scholarpedia.org/article/Reinforcement_learning#.28Temporal.29_Credit_Assignment_Problem.

"Wpovell's Blog." *Making a Pong AI with Policy Gradients*, wpovell.net/posts/pg-pong.html.