Connor Thompson

Csavina and Boston

Capstone

4/3/17

Playing Video Games with Machine Learning

**Introduction**officeArt object

For my capstone project, I chose to pursue a project in the computer science subfield of machine learning and even more specifically, reinforcement learning. Machine learning can be described as “a type of artificial intelligence that provides computers with the ability to learn without being explicitly programmed” (techtarget.com). Machine learning is very similar to data science, but in my mind I picture data science as looking through the data by hand whereas with machine learning we make the computer mine the data for us to reveal the same important insights. Reinforcement learning is a specific type of machine learning in which a program is allowed to make actions in a simulated environment where it can then witness and record both the change in the environment and the “reward” received from the action. From this reward, it will then either increase the likelihood of the action reoccurring given a similar environment or decrease it based on if the reward is positive or negative. Biologically, it is analogous to classical conditioning.

For this project, I chose to implement a reinforcement learning agent on the timeless game of Tetris using Python 3.5. For those of you that don’t remember, Tetris is the classic Atari game with falling “tetrominos” - or shapes made of 4 blocks falling down on your screen. The goal is to fill a line full with these blocks and that then “clears” the line and increases your score. The general idea is that the agent will play Tetris and “train” (more on that later) so that it will ultimately receive the most reward (in this case, clearing as many lines as possible in the fewest games). I decided to use Python because I am familiar with it, and I wanted to experiment with Google’s newly released Python Tensorflow library made specifically for machine learning. I also used Python because it works across all platforms - Windows, Mac, Linux, and even Android - and it is a very high-level programming language, making it easy to write and understand the code. I chose to do this project because machine learning is a field that is very interesting to me, and it is definitely something that I would like to pursue both in my post-high school academics and in my career.

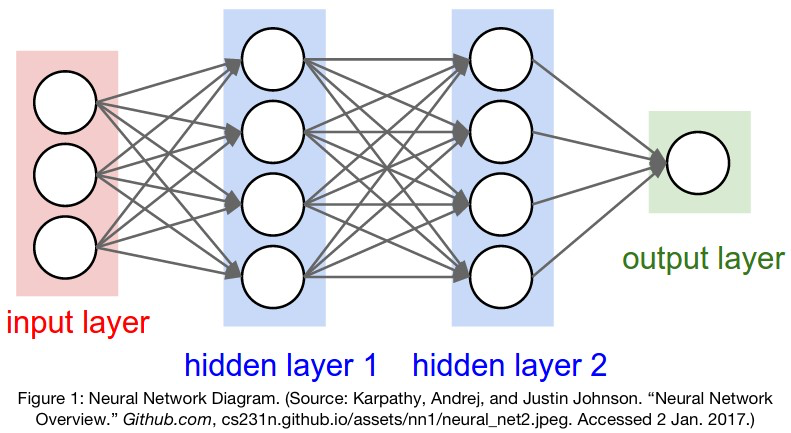
To help prepare for this project, I completed a 12-week Coursera course called “Machine Learning” taught by Stanford's Andrew Ng. This gave me a basic introduction to all the various types of machine learning, but ironically it did not cover reinforcement learning. To help me learn more about reinforcement learning, I used the book *Reinforcement Learning: An Introduction* by Richard Sutton and Andrew Bartoto help understand how to successfully utilize reinforcement learning. These two sources gave me the resources required to work on this project.

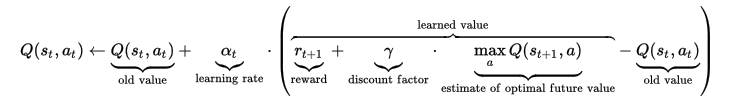
For the end product, I was hoping to create a program that would be able to teach itself to play Tetris, and other games, at an “okay” skill level. At the onset of this project, I had no idea what to expect. Tetris is a very hard game to learn for a computer just because there is so much variation and the largest reward is sparse compared to the rest of the game, making it hard for the program to realize that clearing lines is what gives it the largest reward. I felt reaching an okay skill level was an appropriate goal given the inherent difficulty of the task. For comparison, a group of Stanford students (Stevens) were able to create a version that would score anywhere from 5 - 30 lines per game using more advanced machine learning methods as well as a program designed specifically for Tetris, not playing video games in general as my project is. My project is modeled after Google DeepMind’s original “DQN Paper” published in *Nature* on February 26th 2015 as well as the project DeepLearningVideoGames by Akshay Srivatsan et. al.

**Definitions**officeArt object

Like any computer science paper, there are several definitions we should cover before we dive too deep into the project.

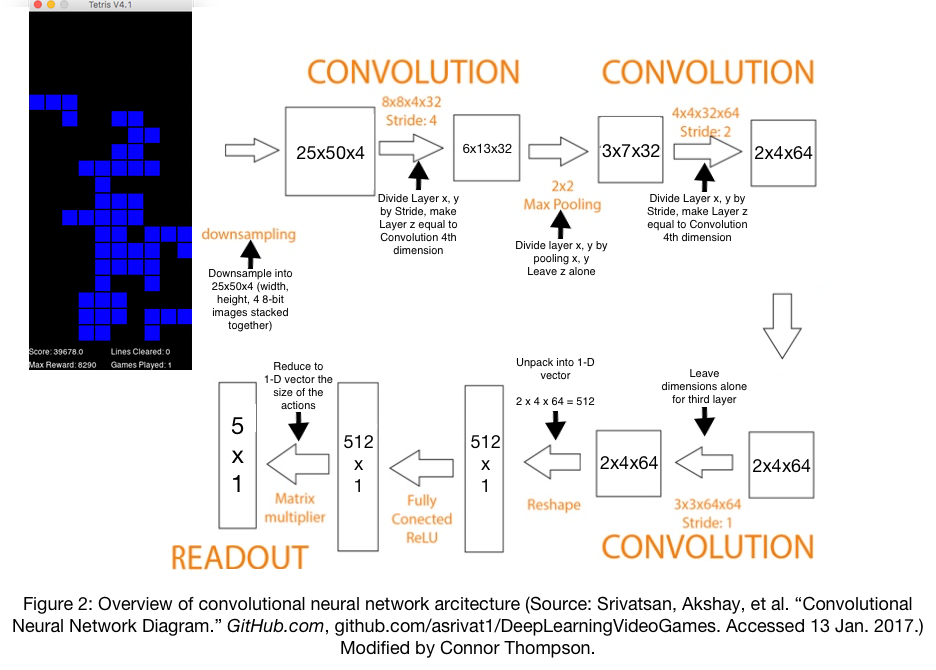
A **convolution** acts as a “sliding window” over our data. They scan our input data in chunks and find reoccurring patterns in them. This allows for pattern recognition and is commonly used for things like face detection in smart phone cameras and Google image searches (this is why Google can differentiate a cat from other animals). Another way to think of a convolution is as a lens through which we can look at our data with each convolution providing us with a different representation of the image.

A **neural network** is a system in computer science where we take inputs (for this project the image data, action taken, and reward received for each frame) and feed them through a series of neurons and layers connected by weights. These neurons will apply operations (in our case convolutions, giving us a **convolutional neural network**) and then give us an output (in our case, the estimated value of each possible action for our given environment). Neural networks try to mimic the biological brain at a smaller scale and as a result are excellent at generalizing their inputs. Neural networks are an old idea (originating as far back as the 1950’s) but they have just recently come into fruition thanks to the large advances in both machine learning techniques (namely deep neural nets, neural networks with many layers and many neurons) and huge advances in computational power (specifically in graphics card processing power and cloud computing). They have revolutionized computing and given way to some of the most amazing innovations to date in products such as IBM’s Watson, Tesla’s self-driving cars, and Google DeepMind’s AlphaGO (more on those later!). 

To be even more specific, the reinforcement learning algorithm that I am employing for this project is **Q-Learning**. When combining Q-Learning with a deep neural network (as in my case) we call the resulting neural network a **Deep Q-Network** (DQN). In the Q-Learning algorithm, we try to estimate the value of an action taken in a state and the future value available within the next state to maximize our total reward. We learn these values by “mining” the data we create by taking these actions and recording their effects. The Q-Learning algorithm is as follows:

where Q(st, at) is the predicted value for action at given state st. We won’t dive into the equation because I don’t want anyone to fall asleep.

Another important aspect of reinforcement learning is the trade-off between exploration and exploitation. To elaborate, if we want to maximize our total score we should **exploit** every action (choose the action with the estimated highest reward *and* future reward) every time. Conversely, **exploring** is taking a random action despite all other input, allowing us to explore the simulated environment. If we simply chose to exploit every action the agent will not be able to learn new ways of getting better rewards faster. In our Tetris example, the agent could learn right away that avoiding a “game over” gives it a small reward, and it could simply keep doing that. However, if it never tries anything new (via exploring), it might not figure out it can clear one/multiple lines and get an even greater reward. To fix this issue, reinforcement learning programs typically start with a high rate of exploration (90%+) and slowly anneal it to a relatively low amount of exploration (anywhere from 0-5%). This allows the agent to initially explore a lot and find good ways to maximize the reward and towards the end it will simply keep exploiting those actions to maximize the reward. Again using our Tetris example, this could mean that it finds out how to clear a line early on and from then on out it will begin to clear more and more lines as our exploration rate decreases since it knows that clearing a line will yield a high reward.

One last term to know is **replay memory**. Replay memory is a "container" for the reinforcement learning agent’s past experiences for use in training the agent. In our case our replay memory will hold an initial frame stack (the 4 frames of the game before the action was taken), the action taken, the reward received from that action (as a whole number), and the new resulting frame stack (with the new, resulting frame “bumping” out the oldest frame). 

**Algorithm**officeArt object

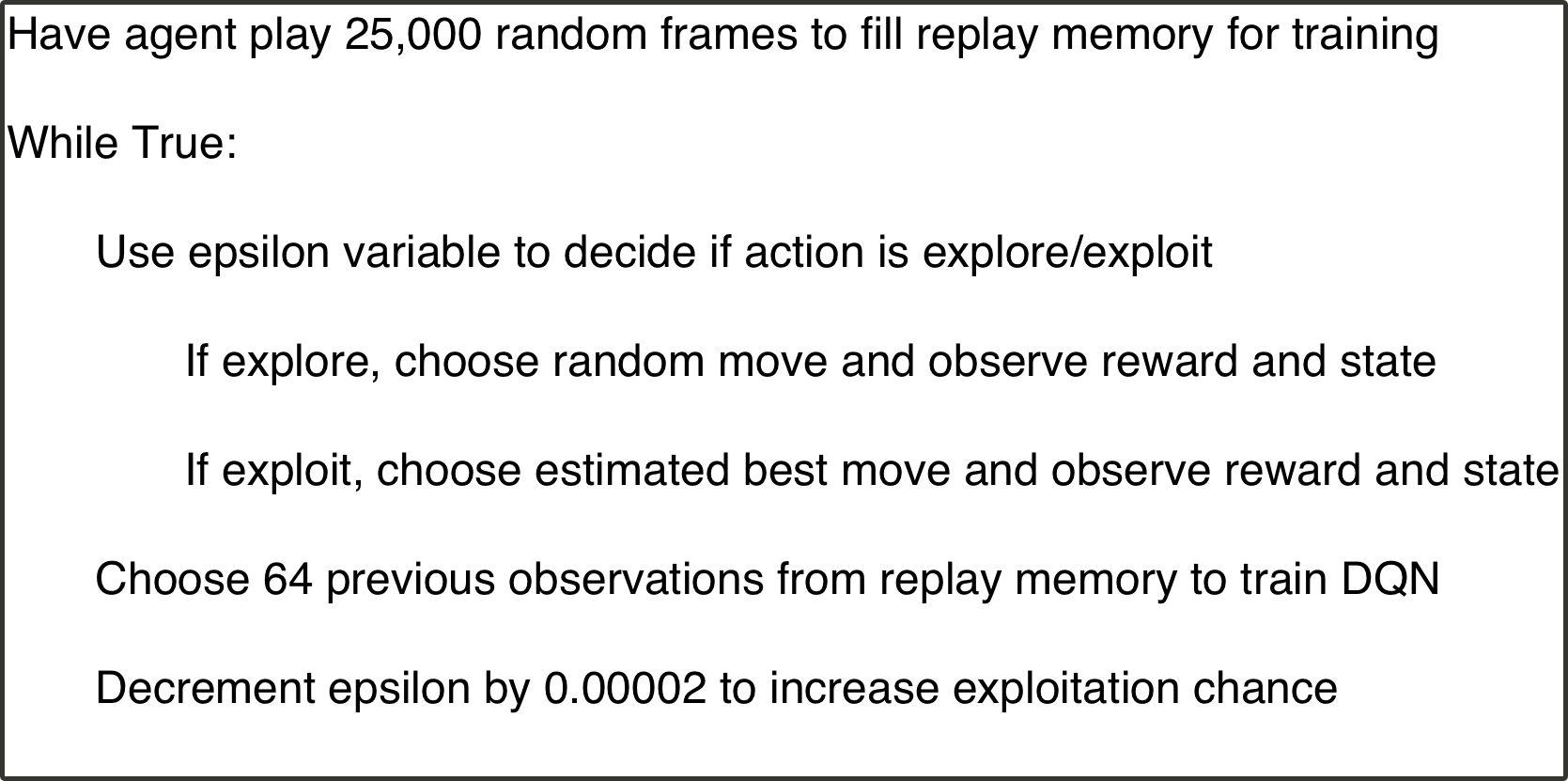
Before we further dive into the findings of this project, it is important to understand how exactly the system works. We start the program by building a convolutional neural network as shown in Figure 2. This convolutional neural network takes the image data from our Tetris game and stacks it into a 4-frame "stack". This gives the neural network the current frame of the game and the three previous frames allowing it to have context about the current environment. Without the frames being stacked, the neural network would not be able to determine what direction the pieces are moving in each frame. This frame stack is then fed through three different convolutions to search for recurring patterns in the data with each convolution looking “deeper” than the previous. Each “layer” in the network is connected by a set of weights that determine how important the output of that previous layer is to the overall output of the current layer. These weights are what allow neural networks to generalize so well as they can be adjusted minute amounts to get ideal performance. Think of it as the unique and ever-changing connections between neurons in the human brain that allow us to learn and adapt to new environments and new ideas. The neural network then outputs the estimated value of each available action (do nothing, rotate left, rotate right, move left, move right). From here, we transition from the normal machine learning portion of my project into the reinforcement learning section where we take the estimated value of each action and either:

A) Exploit the action to receive the largest estimated reward and future reward

or

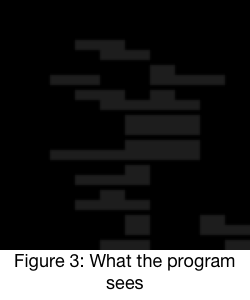
B) Choose a random action and explore, allowing us to potentially discover new ways to maximize our score

We start the game with an absolute chance of exploring (100%), with it gradually annealing down to a 1% chance of exploring. After we take either the exploit/explore action, we then play the action in the game and witness both the changes it makes to the game and the reward we receive from the game. We then append the initial game state, the action chosen, the reward received, and the new game state to our replay memory for later use in training. We then take a random batch of 64 experiences from the replay memory and feed it into the network to calculate the error of the network (by comparing the estimated reward to the actual reward received) and adjust the weights within the convolutional neural network accordingly in order to emphasize or dissuade those actions given a similar environment. This allows the network to learn over time how to maximize its score and is what makes the convolutional neural network (the “architecture”) into a Deep-Q Network (how it is trained). For those technically inclined, I used the Adam algorithm to train the network, a variant of stochastic gradient descent.

**Algorithm**

**Issues**

With such a complex system, I ran into multiple issues during my project. Initially, I had a hard time creating the program. This was my first project (and a rather complex one) with any sort of machine learning, and I jumped into it head first without any idea of where to start. To help me get past this initial "coder's block", I used a reference project (DeepLearningVideoGames) that used the same idea I did to teach a convolutional neural network to play Pong. With the help of this reference project and with my previously accumulated background knowledge, I was able to successfully create the most basic version of my project. Unfortunately, it did not work very well.

The problem with my initial version of the program was that it started off scoring fairly well, with the first version clearing a total of 72 lines over the course of ~12 hours. However, after running the program for a couple of hours, the program would "break" and start sending repeated "do nothing" actions to the game. This was the biggest problem I encountered during the building of this project. Unlike traditional programming, it is very difficult to debug machine learning code. There’s a lot going on and you’re not always sure what the agent is thinking, or in other words, by their very nature machine learning programs are “creative” to a degree. In addition, there are a thousand variables you can play around with that will have varying impacts on your overall success. Some researchers have even had their programs learn to pause themselves in order to maximize their reward! After about a month of on again off again debugging, getting troubleshooting advice from my mentor David Ye, and after trying ~ 5 different reward methods (which is where I initially thought the problem was), I figured out that the program was not reading the game images from its replay memory correctly, stopping it from learning how to play. With this problem now fixed, I had my first version of the program up and running! 

**Findings**officeArt object

The first thing I did to try and improve my program was implementing “prioritized experience replay”, which is the same as the normal experience replay except we train the network based on the experiences with the largest errors in their estimated reward versus their actual received reward. In theory, this would make the network more accurate and allow it to train significantly faster. As expected, implementing this made the estimated values much more accurate but unfortunately the reward method I had settled on by this point was not conducive to the prioritized experience replay and hence could not be used in the end product. The problem in my case was that when the network would score a line, it would strongly weight all its future actions to try and achieve that same state. This sounds ideal, but in my case if a line was cleared by placing a piece on the right side of the board, all the pieces after it would be placed on the right side of the board. There are several ways this problem could be remedied in the future. I personally would investigate reducing the reward for clearing a line or reducing the learning rate of the network further. I could also look into using a “semi-prioritized” experience replay where we train partially based on the experiences with the largest margin of error and partially on random experiences.

The second improvement I made to try and improve the performance was implementing “Doubled Q-Learning” which is the same as Q-Learning but every time we train the network, we have a 50/50 chance of estimating the max value of each action or the minimum value of each action. This allows us to avoid something called “maximization bias” found in traditional reinforcement learning scenarios where the program tends to overestimate the values for each action since it is only estimating the maximum value. In the Q-Learning equation found previously, this is the maxQ parameter. Implementing Doubled Q-Learning in my program improved the performance of the program a noticeable amount although it is hard to provide exact statistics since there are so many variables involved in Tetris and, as a result, each game is entirely unique.

For those technically inclined, I tried using a variety of training methods for the network to see what worked best. All the optimizers had a static learning rate of 0.001 and were used with the default parameters as implemented in Tensorflow v1.0. In the end, I ended up using the Adam algorithm. I also experimented with traditional stochastic gradient descent, Adadelta, and Adagrad with Adagrad also showing decent results in my very limited time frame. If I had more time and resources, I could have further investigated the performance of the various optimizers and how changing their parameters affects learning. Unfortunately, I only had a week or two from the final version of my project to presentation and I wanted to spend as much time as possible training the best network configuration.

Again for those technically inclined, I originally had max pooling layers in between my various convolution layers. In the end I ended up removing all but the first max pooling layer as they were adding computation time and possibly throwing away valuable data due to their downsampling. Removing these layers seemed to make no noticeable impact on performance.

I also tried using several different reward methods (how we calculate the reward for a respective action-state pair) for the program. I tried many different variations, but the two I ended up experimenting with the most was what I will refer to as the “positive reward method” and the “negative reward method”. The positive reward method calculates the reward for each piece individually by just looking at the line(s) the piece landed on and using the sum of the line numbers as the reward. This is hard to picture, but essentially it gave the pieces lower on the board a larger reward (since they were further away from a game over). The negative reward method starts each game with a reward of 1000 and each piece placed on the board subtracts a number from the preceding reward. The equation for this reward method is:

(1 / row number) \* 60

where the lower lines have a larger row number meaning that the lower the line, the lower the reduction in score. Between these two reward methods, I found the positive reward method to be better at avoiding game overs while the negative reward method was better at clearing lines.

Given more time and resources, there are many more experiments I would love to run to see what helps the program achieve optimal performance. The largest problem I ran into by far was lack of time as training the network to a point where I feel performance could be fairly evaluated despite all the entropy took several hours for each adjustment I wanted to evaluate. In the future, I would love to get the prioritized replay working so that this training time could be reduced. I would also love to further evaluate the effects of the different optimizers on the performance of the program as well. In addition, I can think of several promising reward methods that might yield better results such as increasing the reward exponentially based on the number of filled squares in each line so that it would entice the agent to clear the line.

**Real Life Applications** officeArt object

Even though it might seem silly to go through all this work in order to teach a computer to play something as trivial as Tetris, the real life applications of machine learning are far reaching and what we learn from seemingly silly experiments such as this can lead to large advancements in real world applications. Right now, machine learning provides you with shopping recommendations, Netflix recommendations, and things like weather predictions; nothing too special. However, the field is expanding at an insane rate, and the technologies of the future will heavily utilize machine learning. Tesla and Google’s DeepMind are using machine learning to create self-driving cars and programs like AlphaGo, DeepMind’s board game machine learning program that managed to beat one of the top Go players 4-1, an unprecedented event that surprised millions in March 2016. Other technologies, like IBM’s Watson made famous by beating the two Jeopardy “grandmasters”, are powered off machine learning and now have the ability to diagnose patients with more accuracy than a doctor thanks to its ability to read and comprehend more than 7,000 new research papers everyday in conjunction with its training from physicians, patients, textbooks, and everything in between. It has been said that Watson is “beyond an evolutionary step, this is a revolutionary step. This has the potential to totally change the way we talk to medicine.” (Mark Kris). Machine learning can be used for an infinite number of applications ranging from financial trading to cyber security to robotics and everything in between.

**Future**

Machine learning has the vast potential to totally transform our world, and mortality as we know it. The current goal is to use machine learning to create a general artificial intelligence that will be able to be implanted into robots. This will allow for robots to replace certain unpleasant aspects of our lives (think jobs, cleaning, and other “chores”) and leave humans free to pursue whatever we want. This would also enable us to create “superhuman intelligence” within a robot. Companies like Google’s DeepMind are already working on this, and it will be happening sooner rather than later with the current expected time frame being anywhere from 10 - 30 years. With this in mind, humans as a species need to come together to ensure that these worker robots won’t fall solely into the hands of the already rich where they will then be able to monopolize all the money and resources from the common people. While this is more a political and ethical issue than a computer science issue, the main way people are looking at fighting this is by giving a universal basic income to every citizen by taxing either the robots or the goods they make. There is a lot of progress being made in this area, and it will be interesting to see how it develops over time.

Looking even further down the road, many are hoping to be able to transcribe and upload their brain into a robot which will then allow them to essentially escape mortality by leaving their body behind but preserving their mind. This might seem like a far-fetched idea, but various groups at Google are already looking into it and Elon Musk (Founder/CEO of Tesla, SpaceX, PayPal, SolarCity and more) just announced in March 2017 that he would be creating a new company named Neuralink which has essentially this goal. This would obviously revolutionize our world as we know it and allow for future cyborgs to do things such as explore the vast expanses of space without the worry of time or oxygen.

Whether these prospects are terrifying to you or not, it will be interesting to see if we end up in a world full of leisure and exploration unbound by the typical constraints on humans or if we end up in a real life Terminator-esque world where the working machines have rebelled against us. Either way it is happening sooner rather than later and in the end, we will likely not be the only group with a say in the decision.

Bibliography

Asrivat1. “asrivat1/DeepLearningVideoGames.” *GitHub*, 21 June 2016, github.com/asrivat1/DeepLearningVideoGames. Accessed 2 Jan. 2017.

“Q-Learning.” *Wikipedia*, Wikimedia Foundation, 23 Mar. 2017, en.wikipedia.org/wiki/Q-learning. Accessed 5 Jan. 2017.

Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction Second Edition Draft*. 2nd ed., Cambridge, MA, The MIT Press, 2017, [webdocs.cs.ualberta.ca/~sutton/book/bookdraft2016sep.pdf](http://webdocs.cs.ualberta.ca/~sutton/book/bookdraft2016sep.pdf).

“Whatis.techtarget.com.” *Whatis.techtarget.com*, whatis.techtarget.com/definition/machine-learning. Accessed 20 Jan. 2017.

Stevens, Matt, and Sabeek Pradhan. “Reinforcement Learning.” *Reinforcement and Systemic Machine Learning for Decision Making*, doi:10.1002/9781118266502.ch3. Accessed 5 Jan. 2017.