Alex Frazier
HW3

1.1
(a) (4 points) Suppose n = 2 i.e. the input is a pair of binary values. Suppose we have a neural network where we don't have any hidden units and just a single output unit i.e. y = f(w T x + b) is the entire network. What should w, b be if we want to implement boolean AND (i.e. y = 1 only when x = (1, 1)). What about boolean OR?

boolean AND: w = [1, 1], b = -1
boolean OR: w = [1,1], b=0

(b) (1 point) Under the same conditions as above, what boolean function of two variables cannot be represented? You just need to state one, and provide an explanation.

XOR cannot be represented because XOR is not linearly separable and a network with the conditions stated above can only represent linear separable functions.

(c) (4 points) Suppose we now allow a single layer of hidden units i.e. y = f(w T z + b), zj = f(w T j x + bj). Construct a neural network that can implement the boolean function you mentioned previously that could not be represented before. The number of hidden units is up to you, but try to keep it as simple as possible.

We can implement XOR with two hidden units and the following weight matrix.
Weights Hidden = [1, -1; -1, 1], bj = 0, Weights to output = [1 1], b = 0

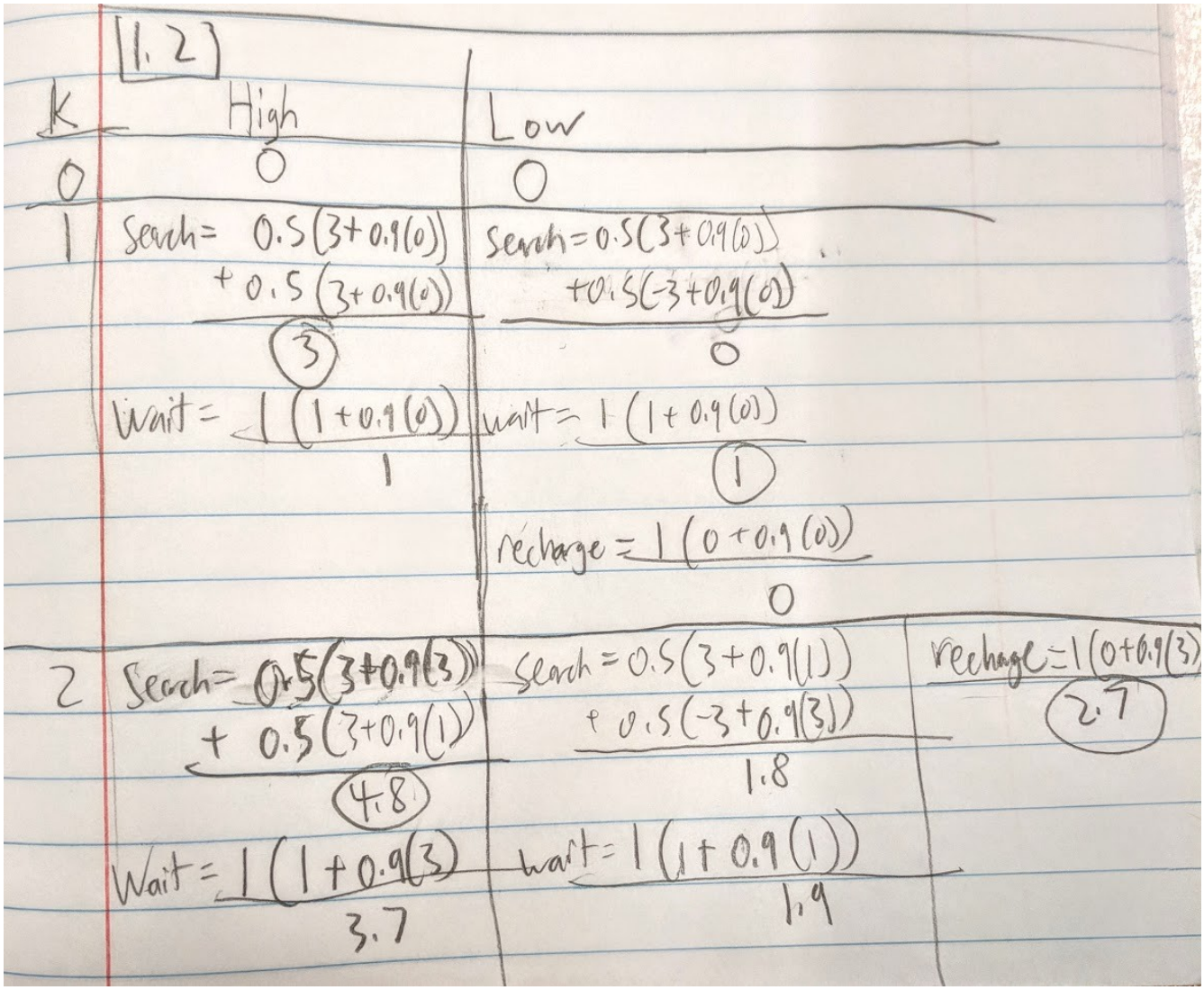
(d) (8 points) It turns out that for any number of input boolean variables, a single hidden layer is enough to represent any boolean function. Describe a general scheme that one can use to construct such a neural network for any boolean function (HINT: consider conjunctive normal form or disjunctive normal form).

Any boolean function can be represented by in conjunctive normal form (as an AND of multiple OR clauses). In part A we showed how to implement a Boolean And and a Boolean OR function with a single unit. If we sum all the variables with bias 0 we can implement the OR of multiple inputs with one unit. If we sum all the variables and set the bias to the negative of the amount one less than the total number of inputs then we can implement the AND of multiple inputs. A general scheme to implement any boolean function would be If we convert our boolean function into CNF form and use the hidden units to implement ORs of the CNF forms OR clauses and then use the output unit to implement an AND of the hidden unit OR clauses.

(e) (3 points) If a single layer is enough to represent all boolean functions, why would you ever want to use multiple hidden layers? What does this suggest about designing deep neural network structures in practice?

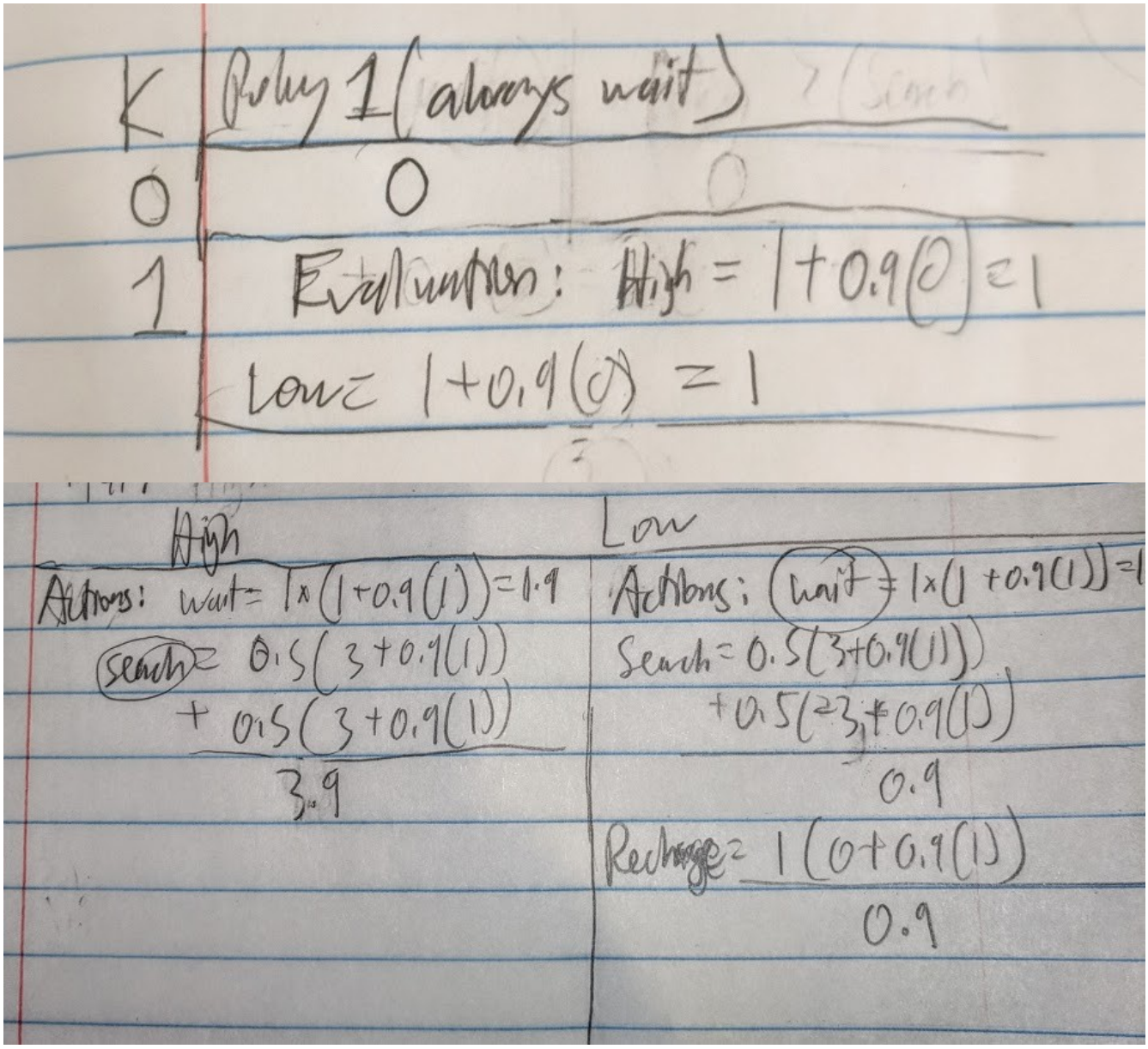
A single layer is enough to represent all boolean functions but may require an exponential number of hidden units to do so. This is because in some cases the CNF form of a Boolean function may require an exponential number of clauses. With multiple layers, earlier layers can extract repeated computations and significantly reduce the number of units needed to represent the function. This means that in practice, using multiple layers are able to represent more information with less units.

1.2
A.



After two iterations of value iteration, the resulting best action values are 4.8 [search] for High and 2.7 [recharge] for Low.

B.



After one iteration of policy iteration we end up with a new policy that if we are in a highly charged state we should search and if we are in a lowly charged state we should wait.

2.7
Report

1. Why mini-batching and why random samples from a larger set of samples to form the mini-batch?

Mini-batching is useful because it allows the network to train on multiple examples at the same time which lets is generalize better to new experiences. The random samples are effective because they allow the network to learn from a group of different experiences during each training. This prevents the network from over training to the same group of examples and also lets the network train on a different group of examples each time. This encourages exploration and gives the network more exposure to different possible Q functions.

Run your code without the implementation of mini-batching, i.e. set K = k = 1. Report what you observe and briefly explain why.

Without the implementation of mini-batching the network is not able to score more than a few points on the game. If we train our Q learning function without mini-batching the network learns from one example out of the batch which may be a very non-relevant example if it was something that happened at the very beginning of the batch. Also, the network cannot generalize the Q function well because it only is trained on one example at a time instead of multiple examples.

You can also run your code with a mini-batch that does not use random sample to construct the batch. Report what you observe and briefly explain why.

If we run the code with a mini-batch that doesnt use random sampling the bird eventually is able to score very high but it during the beginning runs it seems to make the same mistakes each run. This is because the samples that it is training on each team are the same and so it is learning from the same experiences each iteration instead of a random assortment of its past experiences.

2. Suppose your reward model cannot give any hint on the perception, i.e. you can only build your reward model based on crashed, scored. Do you think the algorithm can solve the game? What could be the problem? Briefly explain why.

I do not think that the algorithm could solve the game without the perception. I think that without any perception the algorithm wouldnt be able to compute which state the actor is in and consequently would have to compute Q values only for one state. Then the actor would act in the same way at every state in the game and pick the action with the highest Q value for its one state. Without perception the state part of the Q-update does not exist and the actor cannot differentiate their action decision based on their state.

3. What is the difference between using ReLU and sigmoid as the activation function? How does it apply to this problem?

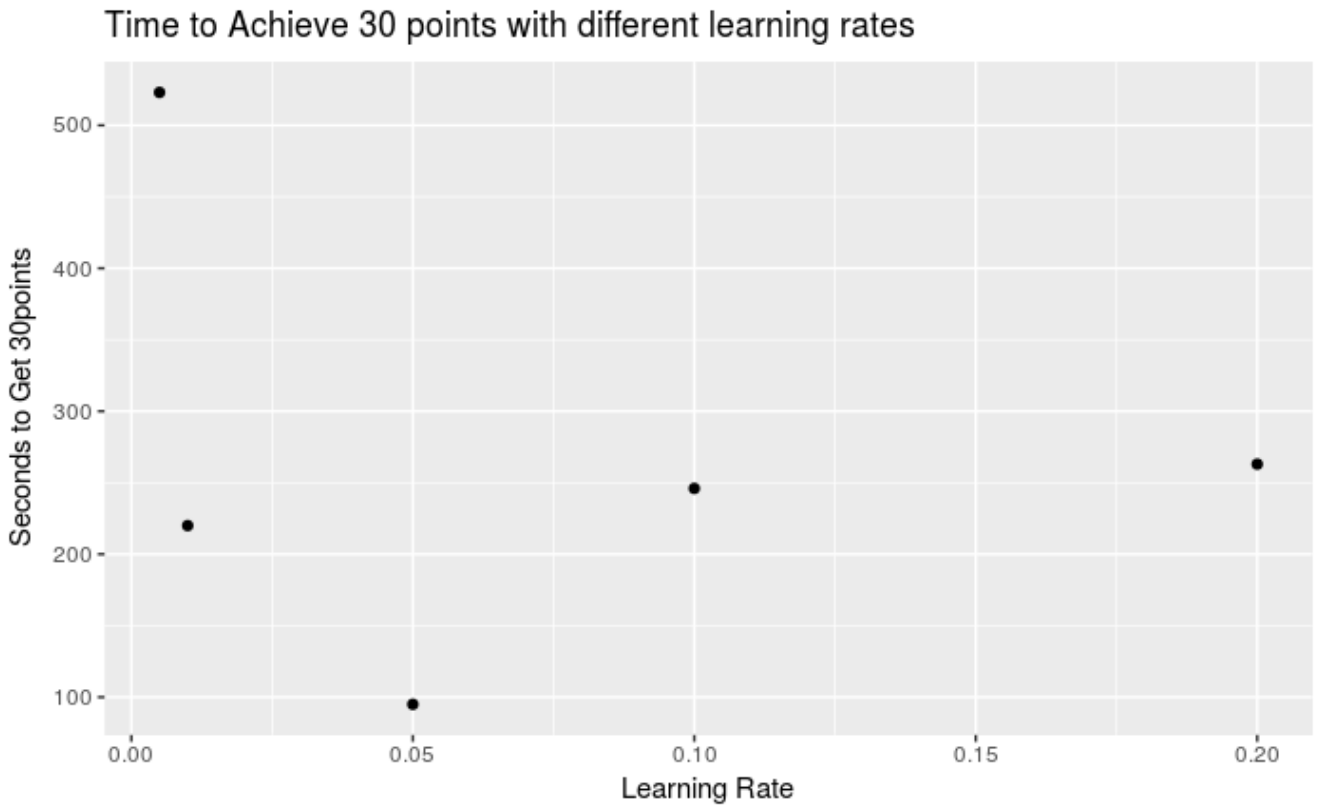
The ReLU activation function computes the output as the max between the input and 0. The sigmoid activation function computes the output as some continuous function of the input that can be positive or negative. The ReLU function tends to cause the network to train faster because it has a constant gradient value as opposed to the sigmoid function which can yield extremely small values and contribute to the vanishing gradient problem. The Relu vs Sigmoid problem applies to this problem

because our keras network used ReLu output units and thus it was able to train at a much faster rate than it would have been able to with sigmoid units.

4. Choose five different learning rates, run your algorithm and record the time for the bird to score 50 for each learning rate (if your solution cannot get to 50, that is OK, use some other number). Plot the graph, list dependent and independent variables, and briefly explain your observations.

Learning Rate	Second to Reach 30 points
0.005	523
0.01	220
0.05	95
0.1	246
0.2	264

Independent variable: Learning Rate
Dependent variable: time to get 30 points



It seems that with a very low learning rate (0.005) the network takes a really long time to learn the optimal Q function (523 seconds) . With too high of a learning rate (0.2) the bird starts to adjust too much and has to go through multiple periods of adjusting and getting a semi high score of around 5-10 but then messing up because of an over adjustment and having to re-learn its way back to 5-10 a few times before getting to the 30 point mark (263 seconds). The sweet spot for my network and reward function setup is a learning rate of about 0.05 that learns the optimal q function fairly fast but doesnt

over learn from examples and usually once it gets past 10 points it will continue on to get 30 points on the same run.