CS230

Learning Nash equilibrium strategies in games Final Report

Sam Ganzfried
sganzfri@stanford.edu

1 Introduction

The central solution concept in game theory is *Nash equilibrium*. Computing a Nash equilibrium can be done in polynomial time in two-player zero-sum games, while it is PPAD-hard in multiplayer and non-zero-sum games and widely believed that no efficient algorithms exist [1]. Several algorithms have been developed for computing Nash equilibrium with varying degrees of accuracy and scalability. In this project I will explore the problem of approximating Nash equilibrium strategies in a game by learning them with a neural network trained on a set of solutions to other games. This provides an alternative to the traditional approach of running an equilibrium-finding algorithm directly on the game at hand. If successful, this has potential to speed up Nash equilibrium approximation if one is able to precompute a database of solutions; for all subsequent games of interest one would just apply the neural network, which could be significantly faster than running an algorithm directly.

A strategic-form game consists of a finite set of players $N = \{1, \ldots, n\}$, a finite set of pure strategies S_i for each player $i \in N$, and a real-valued utility for each player for each strategy vector (aka *strategy profile*), $u_i : S_1 \times \ldots \times S_n \rightarrow \mathbb{R}$. A *mixed strategy* σ_i for player i is a probability distribution over pure strategies, where $\sigma_i(s_{i'})$ is the probability that player i plays pure strategy $s_{i'} \in S_i$ under σ_i. Let Σ_i denote the full set of mixed strategies for player i. A strategy profile $\sigma^* = (\sigma_1^*, \ldots, \sigma_n^*)$ is a *Nash equilibrium* if $u_i(\sigma_i^*, \sigma_{-i}^*) \geq u_i(\sigma_i, \sigma_{-i}^*)$ for all $\sigma_i \in \Sigma_i$ for all $i \in N$, where σ_{-i}^* denotes the vector of the components of strategy σ^* for all players excluding player i. One noteworthy property is that linear transformations of the utilities preserve Nash equilibria. So we can convert any game into a game with all payoffs in [0,1] that has the same Nash equilibria as the original game. For a given candidate strategy profile σ^*, define

$$\epsilon = \epsilon(\sigma^*) = \max_i \max_{\sigma_i \in \Sigma_i} \left(u_i(\sigma_i, \sigma_{-i}^*) - u_i(\sigma_i^*, \sigma_{-i}^*) \right). \quad (1)$$

This value is the standard metric for evaluating the quality of a Nash equilibrium approximation [3].

A two-player game is *zero sum* if the sum of the payoffs equals zero for all strategy profiles, i.e., $u_1(s_1, s_2) + u_2(s_1, s_2) = 0$ for all $s_1 \in S_1, s_2 \in S_2$. Two-player zero-sum games are special for several reasons. The first reason is that the Minimax Theorem holds, which states that every two-player zero-sum game contains a value v_i for player i (with $v_2 = -v_1$), and every Nash equilibrium strategy for player i guarantees a worst-case expected value of at least v_i (and that this is the largest worst-case value that can be guaranteed). Additionally, Nash equilibria are exchangeable; if (σ_1, σ_2) and (τ_1, τ_2) are both Nash equilibria, then (σ_1, τ_2) and (τ_1, σ_2) are also Nash equilibria. Neither of these properties hold in general for non-zero-sum and multiplayer games. As described above, two-player zero-sum games also have a computational advantage over other game classes in that a Nash equilibrium can be found in polynomial time (via solving a linear program).

In our experiments we will consider both two-player zero-sum games, and three-player games. For the two-player zero-sum games we consider games with payoffs for player 1 chosen uniformly at random in [-0.5,0.5], and for three players we use games with all payoffs uniformly random in [0,1].

For both game classes we assume that there are 3 pure strategies per player. We generate a dataset consisting of 10,000 games from each of the classes. For each game, we compute Nash equilibrium strategies for all players. For the two-player zero-sum games we solve the standard linear program formulation, and for the three-player games we use the fastest complete algorithm for computing multiplayer Nash equilibrium, which is based on a non-convex quadratic program formulation [2]. In general for a game with n players and m pure strategies per player, there are m^n strategy profiles, so m^n utilities must be specified for each player. So in general nm^n payoffs must be specified for each game, though for two-player zero-sum games we only need to specify player 1's utilities and therefore require just $m^n = m^2$ values. So for our three-player games, we must specify $nm^n = 3 \cdot 3^3 = 81$ payoffs, and for the two-player zero-sum games must specify $m^2 = 9$ payoffs. These constitute the inputs for a given datapoint. The output is a strategy vector for each player, which will have total length mn. For the three-player games the output for the datapoint will be a vector of 9 real numbers, where all of the numbers are non-negative and the first 3, middle 3, and last 3 numbers each sum to 1. Similarly, for the two-player zero-sum games the output is a vector of 6 numbers.

Suppose that we are given a game with inputs X_i and outputs Y_i, and that the output vector resulting from running our network on X_i is $\hat{Y_i}$. Ideally we would like $\hat{Y_i}$ to minimize $\epsilon()$ as described in Equation 1. However, this is problematic within the context of deep learning for several reasons. First this function is not differentiable, so it would be challenging to use this for the cost function within gradient descent. Furthermore, note that ϵ is a function just of X_i and $\hat{Y_i}$ but not Y_i. So if we used this for the cost function we would not be taking advantage of the equilibrium strategies Y_i that have already been computed.

An alternative approach would be to use a p-norm of $Y_i - \hat{Y_i}$:

$$c(\hat{Y_i}) = \left(\sum_j |Y_{i,j} - \hat{Y_{i,j}}|^p \right)^{\frac{1}{p}}$$

It is clear that if the L^p error were 0, then we would also have $\epsilon = 0$, since $\hat{Y_i}$ would correspond to an exact Nash equilibrium.

Now suppose that we simplify our model and instead of using the full strategy vector of equilibrium strategies for all players as the output, the output is just the equilibrium probability for player 1's first pure strategy. If we construct a neural network using this representation, then we could also use the network to obtain a prediction for the strategy probability for player i's j'th pure strategy by permuting the players and/or payoffs. Thus this will significantly reduce the complexity of our model while not compromising accuracy. (Note also that this could give us an approach to obtain many datapoints from a single game solution if we require additional data.) So for our experiments we will use this model, where the number of inputs per datapoint is as described above, but there is just a single real number output (which is a probability between 0 and 1).

Given this simplified model, we would like to construct a cost function that can be a useful proxy for ϵ. We will be using the L1 norm:

$$c(\hat{Y}) = \sum_i |Y_i - \hat{Y_i}|.$$

Theorem 1 justifies this selection, in that it shows that L1 cost approaching zero implies that ϵ also approaches 0.

Theorem 1. *Suppose that G is a game with all payoffs in [0,1].*[1] *Let σ^* be a Nash equilibrium of G and let σ^δ be a strategy profile such that $|\sigma_i^*(s_i) - \sigma_i^\delta(s_i)| < \delta$ for all $i \in N, s_i \in S_i$. Then* $\lim_{\delta \to 0} \epsilon(\sigma^\delta) = \epsilon(\sigma^*)$.

Proof.

$$\left|u_i(\sigma^*) - u_i(\sigma^\delta)\right|$$

$$= \left| \sum_{s_1 \in S_1} \cdots \sum_{s_n \in S_n} u_i(s_1, \ldots, s_n) \prod_{j=1}^{n} \sigma_j^*(s_j) - \sum_{s_1 \in S_1} \cdots \sum_{s_n \in S_n} u_i(s_1, \ldots, s_n) \prod_{j=1}^{n} \sigma_j^\delta(s_j) \right|$$

[1] Note that our zero-sum games have payoffs in [-0.5,0.5]. We could add 0.5 to all payoffs without changing the equilibria. The resulting game would not be zero sum, but it would be *constant sum* (since all payoffs would now sum to the constant 0.5), and constant-sum games have the same properties as zero-sum games.

$$\left| \sum_{s_1 \in S_1} \cdots \sum_{s_n \in S_n} u_i(s_1, \ldots, s_n) \left(\prod_{j=1}^{n} \sigma_j^*(s_j) - \prod_{j=1}^{n} \sigma_j^\delta(s_j) \right) \right|$$

$$= \sum_{s_1 \in S_1} \cdots \sum_{s_n \in S_n} u_i(s_1, \ldots, s_n) \left| \prod_{j=1}^{n} \sigma_j^*(s_j) - \prod_{j=1}^{n} \sigma_j^\delta(s_j) \right|$$

$$\leq \sum_{s_1 \in S_1} \cdots \sum_{s_n \in S_n} u_i(s_1, \ldots, s_n) \left| \prod_{j=1}^{n} \sigma_j^*(s_j) - \prod_{j=1}^{n} (\sigma_j^*(s_j) - \delta) \right|$$

So it is clear that

$$\lim_{\delta \to 0} \left| u_i(\sigma^*) - u_i(\sigma^\delta) \right| = 0.$$

Similarly we can show that for all $\sigma_i \in \Sigma_i$,

$$\lim_{\delta \to 0} \left| u_i(\sigma_i, \sigma_{-i}^*) - u_i(\sigma_i, \sigma_{-i}^\delta) \right| = 0.$$

So by the triangle inequality we have that

$$\lim_{\delta \to 0} \left| \left(u_i(\sigma_i, \sigma_{-i}^*) - u_i(\sigma_i^*, \sigma_{-i}^*) \right) - \left(u_i(\sigma_i, \sigma_{-i}^\delta) - u_i(\sigma_i^\delta, \sigma_{-i}^\delta) \right) \right| = 0.$$

So

$$\lim_{\delta \to 0} \max_i \max_{\sigma_i \in \Sigma_i} \left| \left(u_i(\sigma_i, \sigma_{-i}^*) - u_i(\sigma_i^*, \sigma_{-i}^*) \right) - \left(u_i(\sigma_i, \sigma_{-i}^\delta) - u_i(\sigma_i^\delta, \sigma_{-i}^\delta) \right) \right| = 0.$$

So

$$\lim_{\delta \to 0} \left| \epsilon(\sigma^*) - \epsilon(\sigma^\delta) \right| = 0.$$

□

2 Experiments

I generated 10,000 games for both classes using the approach described. I assign 95% of the data to be training and 5% to be testing. I use a neural network with 8 ReLU layers followed by a sigmoid layer. The ReLU layers have 70, 60, 50, 40, 30, 20, 10, and 5 neurons respectively. I use Xavier normal initialization for all weights and initialize the bias parameters to 0. The final sigmoid layer ensures that the outputs will be in [0,1], since we are predicting a probability that must lie in [0,1]. As described above, the cost function is $\frac{1}{m} \sum_i |Y_i - \hat{Y}_i|$, where m is the total number of datapoints.

I ran experiments using Tensorflow with the Adam optimizer with a learning rate of 0.0005 and a minibatch size of 32. The costs on two-player zero-sum games after 2000 epochs are shown in Figure 1. The final training error is 0.009, and the testing error is 0.057. By contrast, the error using linear regression is 0.244. The smallest error of using a naïve approach that always outputs a constant value appears to be approximately 0.307 (always predicting 0.25). So using regression provides some improvement over the naïve constant approach, and using a neural network provides a significant improvement over using linear regression.

The results for similar experiments on the three-player game dataset are in Figure 2. For these experiments the final training error is 0.062, and the testing error is 0.324. For comparison, the error using linear regression is 0.380, and best constant error is 0.383. So for these games linear regression provides a negligible improvement over the naïve constant approach, and using a neural network provides some improvement over using linear regression though still results in a very large testing error. The large difference between the training and testing error indicates that our model has high variance. To help combat this, we also experimented using regularization. Results using L2 regularization from the tf.keras.regularizers package with parameter 0.001 are given in Figure 3. The final training error is 0.080 and the testing error is 0.364. It is evident from the figures that the regularization does help smooth the variability in error between training epochs (with a small increase in training error), but it appears that this does not extend to a reduction in overall variance since the testing error remains very large.

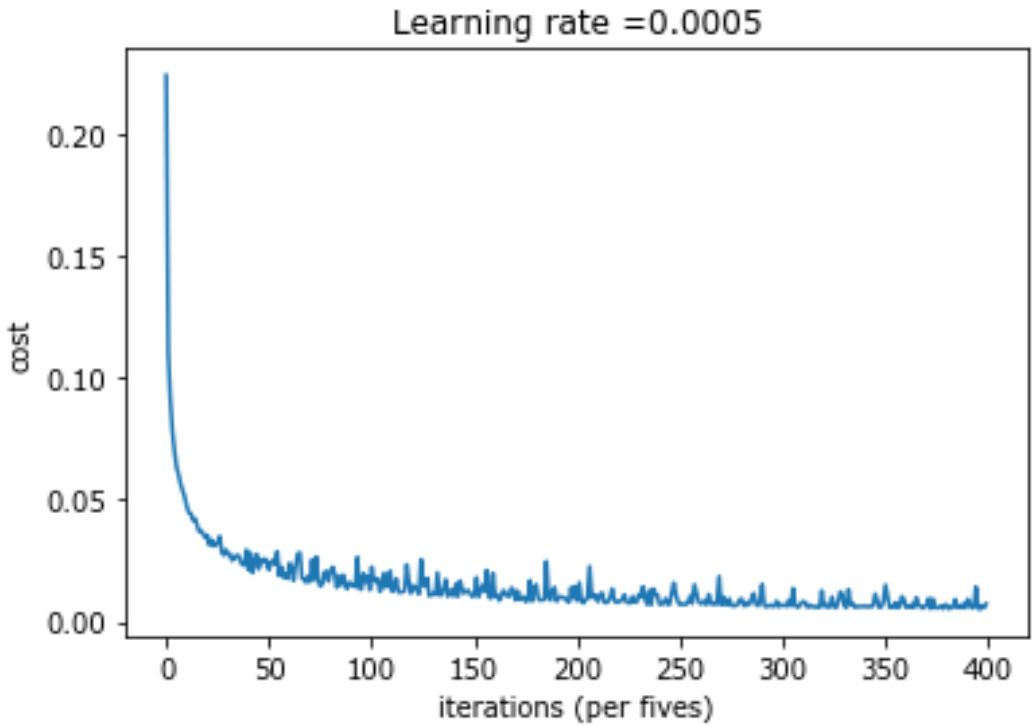

Figure 1: Training error over 2000 epochs on two-player zero-sum games.

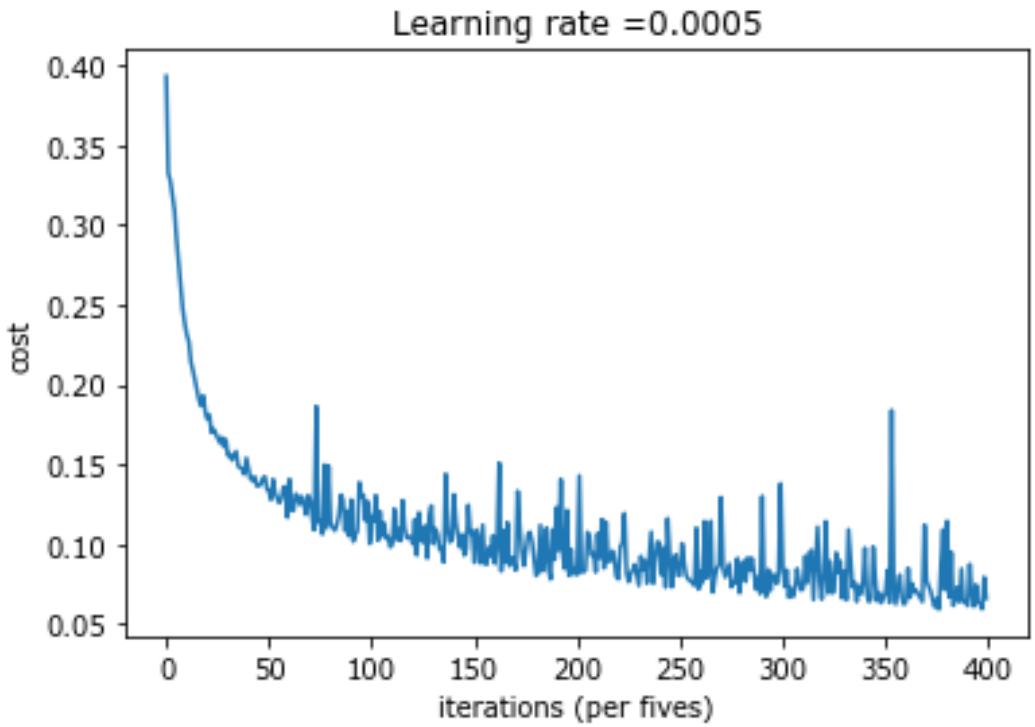

Figure 2: Training error over 2000 epochs on three-player games.

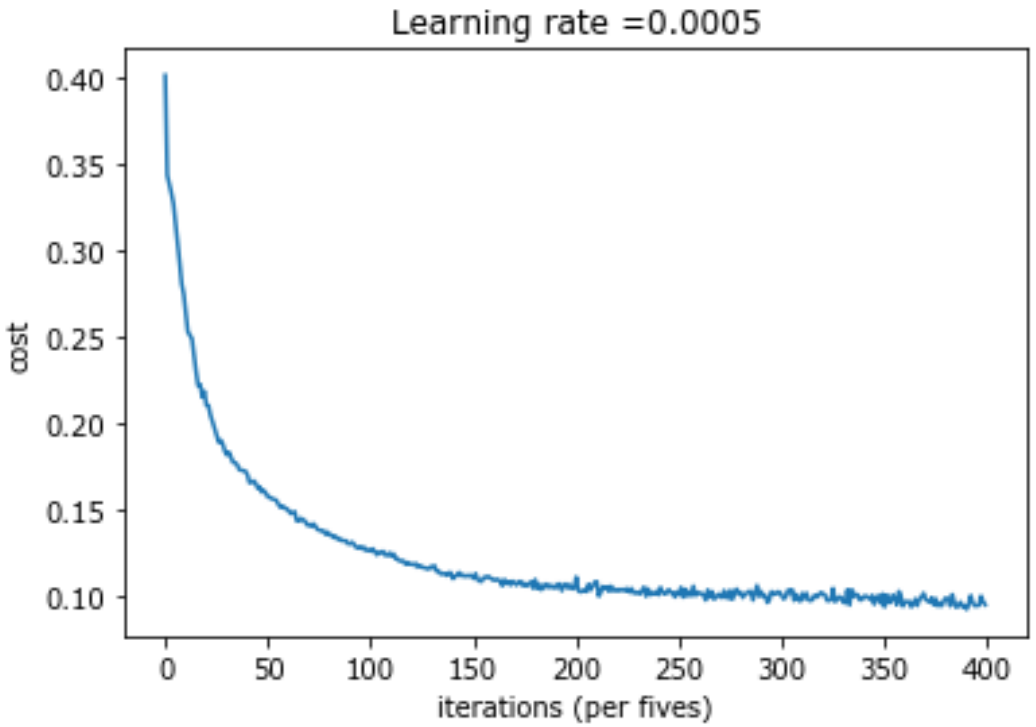

Figure 3: Training error over 2000 epochs on three-player games with regularization.

3 Conclusion

The main conclusion from our experiments is that deep learning appears to be successful for learning Nash equilibrium strategies in two-player zero-sum games, but not for three-player games (though a small improvement is obtained over the baseline regression approach for the latter). This would not be extremely surprising given that several conceptual and computational differences are known to exist between two-player zero-sum and multiplayer games, and the fact that different algorithms were used for the equilibrium computation. Several avenues still remain for improving the three-player results. We can experiment with different forms of regularization, such as dropout, to reduce variance. We can also experiment with larger networks with more layers and more neurons per layer. We have tried using a larger dataset with 100,000 points and this does not seem to improve the testing error.

References

[1] Xi Chen and Xiaotie Deng. 3-Nash is PPAD-complete. *Electronic Colloquium on Computational Complexity*, Report No. 134:1–12, 2005.

[2] Sam Ganzfried. Fast Complete Algorithm for Multiplayer Nash Equilibrium, Feb 2020. arXiv:2002.04734 [cs.GT].

[3] Andrew Gilpin, Javier Peña, and Tuomas Sandholm. First-order algorithm with $\mathcal{O}(\ln(1/\epsilon))$ convergence for ϵ-equilibrium in two-person zero-sum games. *Mathematical Programming*, 133(1–2):279–298, 2012.