Deep Learning for Predicting Human Strategic Behavior

by

Jason Siyanda Hartford

B. Science, University of Witwatersrand, 2008
M. Economics, University of Witwatersrand, 2011

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

October 2016

© Jason Siyanda Hartford, 2016
Abstract

Predicting the behavior of human participants in strategic settings is an important problem for applications that rely on game theoretic reasoning to design mechanisms or allocate resources. Most existing work either assumes that participants are perfectly rational, or attempts to directly model each participant’s cognitive processes based on insights from cognitive psychology and experimental economics. In this work, we present an alternative, a deep learning-based approach that automatically performs cognitive modeling without relying on such expert knowledge. We introduce a novel architecture that allows a single network to generalize across normal form games with varying numbers of actions. We show that the architecture generalists the most successful existing models and that its performance significantly improves upon that of the previous state of the art, which relies on expert-constructed features.
Preface

The work presented in this thesis was performed in collaboration with James R. Wright and my advisor Kevin Leyton-Brown. It has not yet been published, but has been accepted for publication at the Thirtieth Annual Conference on Neural Information Processing Systems, 2016.

I worked with James and Kevin to devise the architecture described in Chapter 3. I wrote all the code used in the experiments described in Chapter 5 with the exception of a software package written by James that presents the data in a manageable format. I wrote the first draft of all the text under the guidance of James and Kevin who later edited and wrote additional text.
# Table of Contents

Abstract ................................................................. ii

Preface ................................................................ iii

Table of Contents ........................................................ iv

List of Tables ................................................................ vi

List of Figures ................................................................ vii

Acknowledgments .............................................................. x

1 Introduction ................................................................. 1

2 Background ................................................................. 4

  2.1 Prediction in Normal Form Games. ............................... 4

      2.1.1 Quantal Response .................................................. 5

      2.1.2 Iterative Strategic Reasoning ................................. 5

  2.2 Deep Learning. .......................................................... 6

      2.2.1 Feed-Forward Neural Networks ............................... 7

      2.2.2 Invariances in Neural Networks .............................. 9

      2.2.3 Neural Networks in Strategic Settings .................... 10

3 Modeling Human Strategic Behavior with Deep Networks .... 11

  3.1 Invariance-Preserving Hidden Units ............................. 13

      3.1.1 Pooling Units ..................................................... 15
### Table of Contents

- **3.1.2 Softmax Output** .................................................. 17
- **3.2 Action Response Layers** ........................................... 18
- **3.3 Output** ................................................................. 20
- **3.4 Relation to Existing Deep Models** ............................. 20

#### 4 Representational Generality of Our Architecture .......................... 22
- **4.1 Behavioral Models** .................................................. 22
  - **4.1.1 Level-\(k\)** .......................................................... 24
  - **4.1.2 Cognitive Hierarchy** ........................................... 24
  - **4.1.3 Quantal Level-\(k\)** ............................................ 25
  - **4.1.4 Quantal Cognitive Hierarchy** ................................ 25
- **4.2 Game Theoretic Features** ........................................ 25

#### 5 Experiments .......................................................................... 27
- **5.1 Experimental Setup** .................................................. 27
- **5.2 Training Details** ........................................................ 28
- **5.3 Experimental Results** ................................................ 29
- **5.4 Regular Neural Network Performance** .............................. 32

#### 6 Discussion and Conclusions .................................................. 34

**Bibliography** ............................................................................. 35

**A Supporting Materials** .......................................................... 39
- **A.1 Representing behavioural features** ................................ 39
  - **A.1.1 Max Max Payoff** .................................................. 40
  - **A.1.2 Max Min Payoff** .................................................. 41
  - **A.1.3 Max Max Efficiency** ............................................. 41
  - **A.1.4 Minimax Regret** .................................................. 42
  - **A.1.5 Min Min Unfairness** ............................................. 42
List of Tables

Table 5.1  Our datasets. Each experiment had subjects play between 8 and 20 games for a total of 128 games, 113 of which were unique.  

28
List of Figures

Figure 1.1  An example $3 \times 3$ normal form game.  

Figure 3.1  This figure shows the direct application of a regular feed forward network to an example $3 \times 3$ game matrix. Two problems with this approach are immediately clear. First, changing the ordering of the actions ($\{T, M, B\}$ and $\{L, C, R\}$) changes the function output; and second if we were to add an action (e.g. add a row to the above payoff matrix), we could no longer apply the network. 

Figure 3.2  A schematic representation of our architecture. The feature layers consist of hidden matrix units (orange), each of which use pooling units to output row- and column-preserving aggregates (blue and purple) before being reduced to distributions over actions in the softmax units (red). Iterative response is modeled using the action response layers (green) and the final output, $y$, is a weighted sum of the row player’s action response layers. 

Figure 3.3  A graphical representation of our hidden matrix units. Each node is a weighted sum of matrices with a single scalar weight associated with each matrix, followed by an element-wise activation function, $\phi(\cdot)$. 
Figure 3.4  The above image shows the input and two hidden layers of a network with a $5 \times 5$ input matrix, three invariance preserving hidden units in the first hidden layer and one unit in the second hidden layer. Without pooling units, each element of every hidden matrix unit depends only on the corresponding elements in the units from the layer below; e.g., the middle element highlighted in red depends only on the value of the elements of the matrices highlighted in orange.

Figure 3.5  This figure shows the same architecture as Figure 3.4 with pooling units added at each layer in the network. Now each element of every hidden matrix unit depends both on the corresponding elements in the units below and the pooled quantity from each row and column. E.g., the light blue and purple blocks represent the row and column-wise aggregates corresponding to their adjacent matrices. The dark blue and purple blocks show which of these values the red element depends on. Thus, the red element depends on both the dark- and light-shaded orange cells.

Figure 3.6  AR layers model best response to beliefs by taking the dot product of weighted sums of preceding AR layers with weighted sums of hidden units. This is equivalent to computing the expected value of an internal representation of utility given by a weighted sum of the hidden units with respect to the distribution defined by the preceding opposition AR layers.

Figure 5.1  Negative Log Likelihood Performance. The error bars represent 95% confidence intervals across 10 rounds of 10-fold cross-validation. We compare various models built using our architecture to a quantal cognitive hierarchy (QCH) model with uniform level-0 behavior (pink line) and QCH with a weighted linear model of level-0 behaviour (blue line).
Figure 5.2  Performance comparison with and without pooling units. All four models were fit with the same hyperparameters using dropout unless otherwise stated, with the only difference being the number of layers and hidden units and whether or not the models used pooling units. ................................. [31]

Figure 5.3  Performance comparison as we vary the number of action response layers. The AR layers appear to result in overfitting in the network as increasing the number of AR layers reduces the average training set loss while increasing the average test set loss. ................................. [32]

Figure 5.4  Performance comparison on $3 \times 3$ games of a feed-forward neural network (FFNet), a feed-forward neural network with data augmentation at every epoch (FFNet (Permuted)), our architecture with two hidden layers of 50 units each (labeled (50, 50)), and quantal cognitive hierarchy with four hand-crafted features (QCH Linear4). ................................. [33]
Acknowledgments

First and foremost, I’d like to thank my advisor Kevin Leyton-Brown for his endless support and patience over the last two years. He has given me more opportunities than I imagined possible and has pushed me to be a better researcher by making me think clearly about what makes a problem interesting.

I also can’t thank James Wright enough for his contributions to this project. James was a second mentor to me, his research provided the solid ground on which this thesis is built, and with his probing and insightful questions he taught me to methodically explore a space of ideas.

Neil, Chris, Alice, Lars and Hedayat the rest of the Game Theory and Decision Theory group provided both helpful research discussions and entertaining lunches when I needed a break from research.

This thesis explores the intersection of game theory and machine learning. Much of my knowledge of machine learning comes from Mark Schmidt and the enlightening discussions at our Machine Learning Reading Group.

Finally, I’d like to thank Nicole for joining me on this crazy journey to the other side of the world. It wouldn’t be the same without you.
For Nicole
Chapter 1

Introduction

Game theory provides a powerful framework for the design and analysis of multi-agent systems that involve strategic interactions [see, e.g., 26]. Prominent examples of such systems include search engines, which use advertising auctions to generate a significant portion of their revenues and rely on game theoretic reasoning to analyze and optimize these mechanisms [10,33]; spectrum auctions, which rely on game theoretic analysis to carefully design the “rules of the game” in order to coordinate the reallocation of valuable radio spectrum [21]; and security systems, which analyze the allocation of security personnel as a game between rational adversaries in order to optimize their use of scarce resources [32]. In such applications, system designers optimize their choices with respect to assumptions about the preferences, beliefs and capabilities of human players [22]. A standard game theoretic approach is to assume that players are perfectly rational expected utility maximizers and, indeed, that they have common knowledge of this. In some applications, such as the high-stakes spectrum auctions just mentioned, this assumption is probably reasonable, as participants are typically large companies that hire consultants to optimize their decision making. In other scenarios that allow less time for planning or involve less sophisticated participants, however, the perfect rationality assumption may lead to suboptimal system designs. For example, Yang et al. [38] were able to improve the performance of systems that defend against adversaries in security games by relaxing the perfect rationality assumption. Of course, relaxing this assumption means finding something else to replace it with: an accurate model of
boundedly rational human behavior.

The behavioral game theory literature has developed a wide range of models for predicting human behavior in strategic settings by incorporating cognitive biases and limitations derived from observations of play and insights from cognitive psychology [2]. Like much previous work, we study the unpeated, simultaneous-move setting, for two reasons. First, the setting is conceptually straightforward: games can be represented in a so-called “normal form”, simply by listing the utilities to each player in for each combination of their actions (e.g., see Figure 1.1). Second, the setting is surprisingly general: auctions, security systems, and many other interactions can be modeled naturally as normal form games. The most successful predictive models for this setting combine notions of iterative reasoning and noisy best response [34] and use hand-crafted features to model the behavior of non-strategic players [36].

The recent success of deep learning has demonstrated that predictive accuracy can often be enhanced, and expert feature engineering dispensed with, by fitting highly flexible models that are capable of learning novel representations. A key feature in successful deep models is the use of careful design choices to encode “basic domain knowledge of the input, in particular its topological structure... to learn better features” [1, emphasis original]. For example, feed-forward neural nets can, in principle, represent the same functions as convolution networks, but the latter tend to be more effective in vision applications because they encode the prior that low-level features should be derived from the pixels within a small neighborhood and that predictions should be invariant to small input translations. Analogously, Clark and Storkey [5] encoded the fact that a Go board is invariant to rotations. These modeling choices constrain more general architectures to a subset of the solution space that is likely to contain good solutions. Our work seeks to do the same for the behavioral game theory setting, identifying novel architectural constraints that extend deep learning to predicting behavior in strategic scenarios encoded as two player, normal-form games.

A key property required of such a model is invariance to game size: a model must be able to take as input an $m \times n$ bimatrix game (i.e., two $m \times n$ matrices encoding the payoffs of players 1 and 2 respectively) and output an $m$-dimensional probability distribution over player 1’s actions, for arbitrary values of $n$ and $m$,
Figure 1.1: An example $3 \times 3$ normal form game. The row player chooses from actions $\{T,M,B\}$ and the column player chooses from actions $\{R,C,L\}$. If the row player played action $T$ and column player played action $C$, their resulting payoffs would be 3 and 5 respectively. Given such a matrix as input we aim to predict the distribution over the row player’s choice of actions defined by the observed frequency of selected actions shown on the right.

Including values that did not appear in training data. In contrast, existing deep models typically assume either a fixed-dimensional input or an arbitrary-length sequence of fixed-dimensional inputs, in both cases with a fixed-dimensional output. We achieve this by leveraging the simplifying behavioral assumption that players are indifferent to the order in which actions are presented.

In Chapter 2, we briefly review the key ideas from the behavioral game theory literature and deep learning upon which this thesis builds, before proceeding to Chapter 3 where we present an architecture that operates on matrices using scalar weights to capture invariance to changes in the size of the input matrices and to permutations of its rows and columns. Chapter 4 shows how this architecture generalizes the models described in Chapter 2. In Chapter 5 we evaluate our model’s ability to predict distributions of play given normal form descriptions of games on a dataset of experimental data from a variety of experiments, and find that our feature-free deep learning model significantly exceeds the performance of the current state-of-the-art model, which has access to hand-tuned features based on expert knowledge [36].
Chapter 2

Background

This work builds on ideas from both the behavioral game theory literature and deep learning. In this chapter, we survey relevant ideas from both fields. Section 2.1 describes key ideas from the behavioral game theory literature that lead to accurate out-of-sample predictions of human behavior in unrepeated normal-form games. Section 2.2 provides a background on deep learning and explains the representation learning philosophy, how successful models encode invariances through modeling choices and prior work applying deep learning to strategic settings.

2.1 Prediction in Normal Form Games.

The task of predicting actions in normal form games has been studied mostly in the behavioral game theory literature. These models tend to have few parameters and to aim to describe previously identified cognitive biases and limitations [2]. While this approach allows researchers to test theories that attempt to explain human behavior in normal-form games, it fails to answer an important practical question: if one wants to use these models to actually predict behavior, which of them provides the best out-of-sample performance? Wright and Leyton-Brown [34, 37] answered this through an extensive meta-analysis comparing the popular approaches used in the behavioral game theory literature. They found that there are two key ideas that support the best performing models: the relaxation of best response to “quantal response” and the notion of “limited iterative strategic reasoning”.

4
We survey these two ideas here, as well as an extension from Wright and Leyton-Brown [36] that improves predictive accuracy by explicitly modeling “non-strategic” players. In doing so, they showed the importance of using hand-crafted behavioral features to improve predictive accuracy. To the best of our knowledge, this approach represents the state-of-the-art method for modeling human behavior in unrepeated normal-form games.

2.1.1 Quantal Response

The first important behavioral observation is that human players tend to play a noisy best response strategy. Models that assume quantal response assume that players select actions with probability increasing in expected utility instead of always selecting the action with the largest expected utility [20]. This is expressed formally by assuming that players select actions, \( a_i \), with probability, \( s_i \), given by the logistic quantal response function,

\[
s_i(a_i) = \frac{\exp(\lambda u_i(a_i, s_{-i}))}{\sum_{a'_i} \exp(\lambda u_i(a'_i, s_{-i}))}.
\]

This function is mathematically equivalent to the “softmax” function (which is widely used in the deep learning literature) with an additional scalar sharpness parameter \( \lambda \) that allows the function to output a strategy that uniformly mixes between the actions in the set of best responses as \( \lambda \to \infty \) (and plays the best response with certainty if it is unique) and the uniform distribution as \( \lambda \to 0 \). This relaxation is motivated by the behavioral notion that if two actions have similar expected utility then they will also have similar probability of being chosen.

2.1.2 Iterative Strategic Reasoning

The second behavioral observation is that players are likely to be boundedly rational in their decision making. Iterative strategic reasoning means that players perform a bounded number of steps of reasoning in deciding on their actions, rather than always converging to fixed points as in classical game theory. Models incorporating this idea typically assume that every agent has an integer level. Non-strategic, level-0 players choose actions uniformly at random; level-\( k \) players...
best respond to the level-\((k-1)\) players in Costa-Gomes et al.’s [8] Level-\(k\) model or to a mixture of levels between level-0 and level-\((k-1)\) in Camerer et al.’s [3] cognitive hierarchy model.

The two ideas can be combined, allowing players to quantally respond to lower level players [30, 35]. We examine the relationship between these models in more detail in Chapter 4.

Non-Strategic Features

The assumption that level-0 players play uniformly at random has an important effect on the performance of the models for two reasons: it has been shown that there tend to be a large number level-0 players [35], and more importantly, the level-0 distribution effectively parameterizes the distributions of play of higher-level players because the models are defined recursively starting from a base case of level-0 behavior.

Given this observation we might expect that the performance of all of the models mentioned above can be improved by better modeling the non-strategic level-0 players. Wright and Leyton-Brown [36] demonstrate this by modeling level-0 behavior using a weighted linear model of hand-crafted “non-strategic features” that each recommend one or more actions. Examples of such features include the action associated with the largest payoff in the matrix, the \(\text{max max payoff}\), or the action that maximizes a player’s worst case payoff, the \(\text{max min payoff}\). These features are combined by taking their linearly weighted sum and normalizing to define a distribution over actions. Their most successful model (in terms of predictive accuracy) combines quantal response and bounded steps of reasoning with the weighted-linear model of non-strategic level-0 behavior. This represents the state-of-the-art behavioral game theory model against which we benchmark our contributions.

2.2 Deep Learning.

The performance of most machine learning algorithms depends heavily on the representation of its input data [13]. The core idea of deep learning is that one should build hierarchical models that may learn more useful and complex representations
at every level of the hierarchy [13]. This approach has had much recent success in solving supervised learning problems in vision, speech and natural language processing [for recent surveys see, 17, 24].

In this section we begin by reviewing feed-forward neural networks and then describe how they have been constrained using domain-specific invariances to improve performance for tasks in particular domains.

2.2.1 Feed-Forward Neural Networks

Feed-forward neural networks (or *multi-layer perceptrons*) are composite functions that map from an input vector in $\mathbb{R}^n$ to an output vector in $\mathbb{R}^m$ through a sequence of linear transformations and element-wise nonlinearities as follows,

$$f(x) = W_n \phi \left( W_{n-1} \phi \left( \ldots \phi \left( W_2 \phi \left( W_1 x + b_1 \right) + b_2 \right) \ldots \right) + b_{n-1} \right) + b_n$$

where $f_i(y) = \phi(W_i y + b_i)$ and $\phi(x)$ is some non-linear activation function applied element-wise. Commonly used non-linearities are the sigmoid function, $\text{sig}(x) = \frac{1}{1+e^{-x}}$, the hyperbolic tangent function, $\text{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, and rectified linear units, $\text{relu}(x) = \max(0,x)$. Each function $f_i$ is referred to as a “layer” in a network, where layers are vector-to-vector maps. Each scalar element of the vector output of $f_i$ is referred to as a “node” or “neuron” in the network.

We can evaluate the performance of the neural network by specifying a loss function that evaluates the output of the neural network for a given input vector with respect to the corresponding target values observed in the data. The loss function, $L(\theta | \{(x_i, y_i)\}_{i=1}^N)$ where $\theta$ is the set of parameters of the model, $(x_i, y_i)$ are the $N$ tuples of training examples from the dataset, is typically derived by maximizing the likelihood of the data with respect to the parameters of the network.

Because each of the functions, $f_i$, is continuous and differentiable (almost) everywhere\(^1\), one can compute gradients of the loss function with respect to each of the parameter vectors/matrices $(W_i, b_i)$ in the network using the chain rule. This operation can be done efficiently using the backpropagation algorithm by noticing

\(^1\text{relu}(x)\) is not differentiable at $x = 0$ but a set of sub-gradients exists.
that because
\[
\frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial f_{i+1}} \frac{\partial f_i}{\partial f_i} \frac{\partial f_i}{\partial \theta_i},
\]
we can reuse previously computed values \( \frac{\partial L}{\partial f_{i+1}} \) if we compute gradients “backwards” from the output layer to the input layer. For each layer \( i \), we only have to compute \( \frac{\partial f_{i+1}}{\partial f_i} \frac{\partial f_i}{\partial \theta_i} \) and can reuse computation from the previous layer to get \( \frac{\partial L}{\partial f_{i+1}} \). This procedure allows us to compute gradients with respect to all parameters in the network in time proportional to the forward pass through the network. Given these gradients, the parameters of the network can be optimized using stochastic gradient descent.

Neural networks are non-convex so gradient decent is not guaranteed to reach the global optimum, but empirically this tends not to be a problem as researchers have found that for the high-dimensional models used in practice, most local optima tend to offer similar generalization performance [4, 9].

A bigger challenge is overfitting: neural networks are very flexible models, which makes them prone to overfitting the training data and poor generalization performance. Researchers combat this using a selection of regularization techniques. The simplest of these limits the capacity of the model by adding a norm penalty term to the loss function as follows,
\[
\tilde{L}(\theta|\{(x_i, y_i)\}_{i=1}^N) = L(\theta|\{(x_i, y_i)\}_{i=1}^N) + \alpha \|\theta\|_{j \in \{1, 2\}}
\]
Both the the absolute value norm \( (j = 1, \text{ known as } L_1 \text{ regularization}) \) and Euclidean norm \( (j = 2, \text{ known as } L_2 \text{ regularization or weight decay}) \) are commonly used in practice. The former tends to result in more sparse solutions, while the latter penalizes larger weight values more than smaller values and hence results in smaller average parameter values.

An alternative approach to avoiding overfitting is model averaging. Averaging the output of a large number of neural networks tends to outperform any individual model because while individual networks that overfit tend to exhibit high variance, they are not necessarily biased and hence averaging their predictions reduces variance thereby improving generalization performance. Unfortunately, building large ensembles is infeasible because each network takes too long to train.
Dropout [28] is an alternative approach that can be interpreted as an efficient form of model averaging. It works by “dropping out” a different random subset of the nodes in the network at every mini-batch of training by element-wise multiplying the output of each $f_i$ by a random binary mask. At test time, one can either take a Monte-Carlo average of the model’s predictions by sampling a large number of binary masks and averaging the resulting prediction, or one can approximate the Monte-Carlo average by removing the binary mask and scaling the outputs of the nodes by the probability that they are retained.

2.2.2 Invariances in Neural Networks

While a sufficiently large feed forward network could, in theory, represent the same functions as those used to achieve most state-of-the-art results published in the literature\(^2\) in practice they tend to be outperformed by models that have a more constrained output space that limits the network to solutions that are “reasonable” in a given domain.

Convolutional neural networks are the best example of this approach. Instead of applying an arbitrary affine transformation at every layer of the network, convolution networks instead apply multiple convolution filters at each layer and the network is parameterized by the weights of their convolution kernels. The convolution operation is linear so there exists an equivalent affine transformation, but it requires a far larger number of parameters and involves significant redundancy because the affine transformation representation has a large number of weights tied to the same value. By constraining the network in this fashion we encode the fact that natural images tend to be invariant to translation. In some domains we can take this idea further. For example, Clark and Storkey [5] encode the rotational invariance of the playing surface in the board game Go by constraining the convolution kernels to be symmetric.

We would like to predict action distributions in normal-form games with an arbitrary number of actions. This requires invariance to the size of the network’s input. The best known prior work that achieves an analogous goal is Long et al. [19]’s Fully Convolutional Network architecture which predicts a class label for

\(^2\)For domains that depend on sequences, recurrent connections are also necessary. We omit a discussion of recurrent neural networks here because they are not relevant to this work.
each pixel in an arbitrarily sized image. They achieve this by replacing the “fully connected layers” (regular feed-forward layers) that are typically used in the final layers of a convolutional network with convolutions that map to a filter for each class label.

While regular convolutional structure is useful in images, our model is most mathematically similar to “MLP Convolutions” developed in Lin et al. [18]’s Network in Network model, though we derived our architecture independently using game theoretic invariances. We discuss the relationships between the two models in Section 3.4 at the end of Chapter 3.

2.2.3 Neural Networks in Strategic Settings

There have been relatively few applications of deep learning to multiagent settings. Notable exceptions are the evaluation function in Clark and Storkey [5] and the policy network used in Silver et al. [27]’s work in predicting the actions of human players in Go. Their approach is similar in spirit to ours: they map from a description of the Go board at every move to the choices made by human players, while we perform the same mapping from a normal form game. The setting differs in that Go is a single, sequential, zero-sum game with a far larger, but fixed, action space, which requires an architecture tailored for pattern recognition on the Go board. In contrast, we focus on constructing an architecture that generalizes across general-sum, normal form games.
Chapter 3

Modeling Human Strategic Behavior with Deep Networks

In this chapter we explore how one might use deep learning to model human behavior in normal form games. A natural starting point in applying deep networks to a new domain is testing the performance of a regular feed-forward neural network. To apply such a model to a normal form game, we need to flatten the utility values into a single vector of length $mn + nm$ and learn a function that maps to the $m$-simplex output via multiple hidden layers. Feed-forward networks can’t handle size-invariant inputs, but we can temporarily set that problem aside by restricting ourselves to games with a fixed input size. We experimented with that approach and found that feed-forward networks often generalized poorly as the network overfitted the training data (see Section 5.4 of Chapter 5). One way of combating overfitting is to encourage invariance through data augmentation: for example, one may augment a dataset of images by rotating, shifting and scaling the images slightly. In games, a natural simplifying assumption is that players are indifferent to the order in which actions are presented, implying invariance to permutations of the payoff matrix.\footnote{We thus ignore salience effects that could arise from action ordering; we plan to explore this in future work.} Incorporating this assumption by randomly permuting rows or columns of the payoff matrix at every epoch of training dramatically improved the generalization performance of a feed-forward network in our experiments, but the network is...
Our approach is to enforce this invariance in the model architecture rather than through data augmentation. We then add further flexibility using novel “pooling units” and by incorporating iterative response ideas inspired by behavioral game theory models. The result is a model that is flexible enough to represent the all the models surveyed in Wright and Leyton-Brown [35, 36]—and a huge space of novel models as well—and which can be identified automatically. The model is also invariant to the size of the input payoff matrix, differentiable end to end and trainable using standard gradient-based optimization.

The model has two parts: feature layers and action response layers; see Figure 3.2 for a graphical overview. The feature layers take the row and column player’s normalized utility matrices $U^{(r)}$ and $U^{(c)} \in \mathbb{R}^{m \times n}$ as input, where the row player has
actions and the column player has \( n \) actions. The feature layers consist of multiple levels of hidden matrix units, \( H_{i,j}^{(r)} \in \mathbb{R}^{m \times n} \), each of which calculates a weighted sum of the units below and applies a non-linear activation function. Each layer of hidden units is followed by pooling units, which output aggregated versions of the hidden matrices to be used by the following layer. After multiple layers, the matrices are aggregated to vectors and normalized to a distribution over actions, \( f_i^{(r)} \in \Delta^m \) in softmax units. We refer to these distributions as features because they encode higher-level representations of the input matrices that may be combined to construct the output distribution. The rest of the model is a cognitive hierarchy model, and so these can be interpreted as the L0 features in such an architecture.

The behavioral game theory literature has shown that iterative strategic reasoning is an important phenomenon in human decision making (see Section 2.1.2 of Chapter 2 for details); we thus want to allow our models the option of incorporating such reasoning. To do so, we compute features for the column player in the same manner by applying the feature layers to the transpose of the input matrices, which outputs \( f_i^{(c)} \in \Delta^n \). The first action response layer for each player is built from a weighted sum of their feature layer outputs. Each subsequent action response layer for a given player then takes the opposite player’s preceding action response layers as input and uses them to construct distributions over the respective players’ outputs. The final output \( y \in \Delta^m \) is a weighted sum of all action response layers’ outputs.

### 3.1 Invariance-Preserving Hidden Units

We build a model that ties parameters in our network by encoding the assumption that players reason about each action identically. Our assumption that the row player evaluates each action in the same way implies that she applies the same function to each row in the utility matrices. Thus, in a normal form game represented by the utility matrices \( U^{(r)} \) and \( U^{(c)} \), the weights associated with each row of \( U^{(r)} \) and \( U^{(c)} \) must be the same. Similarly, the corresponding assumption about the column player implies that the weights associated with each column of \( U^{(r)} \) and \( U^{(c)} \) must also be the same. We can satisfy both assumptions by applying a single scalar weight to each of the utility matrices, computing \( w_r U^{(r)} + w_c U^{(c)} \). This idea
can be generalized as in a standard feed-forward network to allow us to fit more complex functions. A hidden matrix unit taking all the preceding hidden matrix units as input can be calculated as

\[ H_{l,i} = \phi \left( \sum_j w_{l,i,j} H_{l-1,j} + b_{l,i} \right) \quad H_{l,i} \in \mathbb{R}^{m \times n}, \]

where \( H_{l,i} \) is the \( i^{th} \) hidden unit matrix for layer \( l \), \( w_{l,i,j} \) is the \( j^{th} \) scalar weight, \( b_{l,i} \) is a scalar bias variable, and \( \phi \) is a non-linear activation function applied element-wise (see Figure 3.3 for a graphical representation).

Notice that, as in a traditional feed-forward neural network, the output of each hidden unit is simply a nonlinear transformation of the weighted sum of the preceding layer’s hidden units. Our architecture differs by maintaining a matrix at each hidden unit instead of a scalar. So while in a traditional feed-forward network
each hidden unit maps the previous layer’s vector of outputs into a scalar output, in our architecture each hidden unit maps a tensor of outputs from the previous layer into a matrix output.

Tying weights in this way reduces the number of parameters in our network by a factor of $nm$, offering two benefits. First, it reduces the degree to which the network is able to overfit; second and more importantly, it makes the model invariant to the size of the input matrices. To see this, notice that each hidden unit maps from a tensor containing the $k$ output matrices of the preceding layer in $\mathbb{R}^{k \times m \times n}$ to a matrix in $\mathbb{R}^{m \times n}$ using $k$ weights. Thus our number of parameters in each layer depends on the number of hidden units in the preceding layer, but not on the sizes of the input and output matrices. This allows the model to generalize to input sizes that do not appear in training data.

### 3.1.1 Pooling Units

A limitation of the weight tying used in our hidden matrix units is that it forces independence between the elements of their matrices, preventing the network from learning functions that compare the values of related elements (see Figure 3.5 (left)). Recall that each element of the matrices in our model corresponds to an outcome in a normal form game. A natural game theoretic notion of the “related elements” which we’d like our model to be able to compare is the set of payoffs associated with each of the players’ actions that led to that outcome. This corresponds to the row and column of each matrix associated with the particular element.
This observation motivates our pooling units, which allow information sharing by outputting aggregated versions of their input matrix that may be used by later layers in the network to learn to compare the values of a particular cell in a matrix and its row- or column-wise aggregates.

\[
H \rightarrow \{H_c, H_r\} = \left\{ \begin{pmatrix} \max_i h_{i,1} & \max_i h_{i,2} & \cdots \\ \max_i h_{i,1} & \max_i h_{i,2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}, \begin{pmatrix} \max_j h_{1,j} & \max_j h_{1,j} & \cdots \\ \max_j h_{2,j} & \max_j h_{2,j} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix} \right\} \tag{3.1}
\]

A pooling unit takes a matrix as input and outputs two matrices constructed from row- and column-preserving pooling operations respectively. In principle, a pooling operation could be any continuous function that maps from \( \mathbb{R}^n \rightarrow \mathbb{R} \), but some functions stand out as more behaviorally plausible. One might imagine players comparing payoffs to the best possible payoff associated with an action, implying the \( \max \) pooling operation, or to the average payoffs associated with an action, implying the \( \text{mean} \) pooling operation. In practice we found the \( \max \) function offered the best performance.

Equation (3.1) shows the output of a \( \max \) pooling unit applied to some arbitrary matrix \( H \). The first of the two outputs, \( H_c \), is column-preserving in that it selects the maximum value in each column of \( H \) and then stacks the resulting vector \( n \)-dimensional vector \( m \) times such that the dimensionality of \( H \) and \( H_c \) are the same. Similarly, the row-preserving output constructs a vector of the max elements in each column and stacks the resulting \( m \)-dimensional vector \( n \) times such that \( H_r \) and \( H \) have the same dimensionality. We stack the vectors that result from the pooling operation in this fashion so that at the hidden units from the next layer in the network may take \( H, H_c \) and \( H_r \) as input. This allows these later hidden units to learn functions where each element of their output is a function both of the corresponding element from the matrices below as well as their row and column-preserving maximums (compare Figure 3.4 with Figure 3.5).
3.1.2 Softmax Output

Our model predicts a distribution over the row player’s actions. In order to do this, we need to map from the hidden matrices in the final layer, \( H_{L,i} \in R^{n \times n} \), of the network onto a point on the \( m \)-simplex, \( \Delta^m \). We achieve this mapping by applying a row-preserving sum to each of the final layer hidden matrices \( H_{L,i} \) (i.e. we sum uniformly over the columns of the matrix as described above) and then applying a softmax function to convert each of the resulting vectors \( h_i \) into normalized distributions. This produces \( k \) features \( f_i \), each of which is a distribution over the row player’s \( m \) actions:

\[
\begin{align*}
f_i &= \text{softmax} \left( h_i^{(i)} \right) \quad \text{where} \quad h_i^{(i)} = \sum_{k=1}^{n} h_{j,k}^{(i)} \text{ for all } j \in \{1, \ldots, m\}, \quad h_{j,k}^{(i)} \in H^{(i)} \quad i \in \{1, \ldots, k\}.
\end{align*}
\]

We can then produce the output of our features, \( \text{ar}_0 \), using a weighted sum of the individual features, \( \text{ar}_0 = \sum_{i=1}^{k} w_i f_i \), where we optimize \( w_i \) under simplex constraints, \( w_i \geq 0 \), \( \sum_i w_i = 1 \). Because each \( f_i \) is a distribution and our weights \( w_i \) are points on the simplex, the output of the feature layers is a mixture of distributions.
Figure 3.5: This figure shows the same architecture as Figure 3.4 with pooling units added at each layer in the network. Now each element of every hidden matrix unit depends both on the corresponding elements in the units below and the pooled quantity from each row and column. E.g., the light blue and purple blocks represent the row and column-wise aggregates corresponding to their adjacent matrices. The dark blue and purple blocks show which of these values the red element depends on. Thus, the red element depends on both the dark- and light-shaded orange cells.

3.2 Action Response Layers

The feature layers described above are sufficient to meet our objective of mapping from the input payoff matrices to a distribution over the row player’s actions. However, this architecture is not capable of explicitly representing iterative strategic reasoning, which the behavioral game theory literature has identified as an important modeling ingredient. We incorporate this ingredient using action response layers: the first player can respond to the second’s beliefs, the second can respond to this response by the first player, and so on to some finite depth. The proportion of players in the population who iterate at each depth is a parameter of the model; thus, our architecture is also able to learn not to perform iterative reasoning.

More formally, we begin by denoting the output of the feature layers as $\mathbf{ar}_0^{(r)} = \sum_{i=1}^k w_i^{(r)} f_i^{(r)}$, where we now include an index $(r)$ to refer to the output of row player’s action response layer $\mathbf{ar}_0^{(r)} \in \Delta^m$. Similarly, by applying the feature layers to a transposed version of the input matrices, the model also outputs a corresponding $\mathbf{ar}_0^{(c)} \in \Delta^n$ for the column player which expresses the row player’s beliefs about
which actions the column player will choose. Each action response layer composes its output by calculating the expected value of an internal representation of utility with respect to its belief distribution over the opposition actions (see Figure 3.6). For this internal representation of utility we chose a weighted sum of the final layer of the hidden layers, $\sum_j w_j H_{L,i}$, because each $H_{L,i}$ is already some non-linear transformation of the original payoff matrix, and so this allows the model to express utility as a transformation of the original payoffs. Given the matrix that results from this sum, we can compute expected utility with respect to the vector of beliefs about the opposition’s choice of actions, $\mathbf{ar}_j^{(c)}$, by simply taking the dot product of the weighted sum and beliefs. When we iterate this process of responding to beliefs about one’s opposition more than once, higher-level players will respond to beliefs, $\mathbf{ar}_i$, for all $i$ less than their level and then output a weighted combination of these responses using some weights, $v_{l,i}$. Putting this together, the

\[ \mathbf{ar}_{k+1} = f \left( \sum_j w_j H_{l,j} \right) \]

where $f(\cdot) = \text{softmax}(x)$.
The $l^{th}$ action response layer for the row player ($r$) is defined as

$$\mathbf{ar}_l^{(r)} = \text{softmax} \left( \lambda_l \left( \sum_{j=0}^{l-1} v_{l,j}^{(r)} \sum_{i=1}^{k} w_{i,j}^{(r)} \mathbf{H}_{L,i}^{(r)} \right) \cdot \mathbf{ar}_j^{(c)} \right), \quad \mathbf{ar}_l^{(r)} \in \Delta^m, l \in \{1, \ldots, K\},$$

where $l$ indexes the action response layer, $\lambda_l$ is a scalar sharpness parameter that allows us to sharpen the resulting distribution, $w_{i,j}^{(r)}$ and $v_{l,j}^{(r)}$ are scalar weights, $\mathbf{H}_{L,i}^{(r)}$ are the row player's $k$ hidden units from the final hidden layer $L$, $\mathbf{ar}_j^{(c)}$ is the output of the column player's $j^{th}$ action response layer, and $K$ is the total number of action response layers. We constrain $w_{i,j}^{(r)}$ and $v_{l,j}^{(r)}$ to the simplex and use $\lambda_l$ to sharpen the output distribution so that we can optimize the sharpness of the distribution and relative weighting of its terms independently. We build up the column player’s action response layer, $\mathbf{ar}_j^{(c)}$, similarly, using the column player’s internal utility representation, $\mathbf{H}_{L,j}^{(c)}$, responding to the row player’s action response layers, $\mathbf{ar}_j^{(r)}$. These layers are not used in the final output directly but are relied upon by subsequent action response layers of the row player.

### 3.3 Output

Our model’s final output is a weighted sum of the outputs of the row player’s action response layers. This output needs to be a valid distribution over actions. Because each of the action response layers also outputs a distribution over actions, we can achieve this requirement by constraining these weights to the simplex, thereby ensuring that the output is just a mixture of distributions. The model’s output is thus

$$\mathbf{y} = \sum_{j=1}^{K} w_j \mathbf{ar}_j^{(r)}, \text{ where } \mathbf{y} \text{ and } \mathbf{ar}_j^{(r)} \in \Delta^m, \text{ and } w_j \in \Delta^K.$$  

### 3.4 Relation to Existing Deep Models

Our model’s functional form has interesting connections with existing deep model architectures. We discuss two of these connections here. First, our invariance-preserving hidden layers can be encoded as **MLP Convolution Layers** described in Lin et al. [18] with the two-channel $1 \times 1$ input $x_{i,j}$ corresponding to the two players’ respective payoffs when actions $i$ and $j$ are played (using patches larger than $1 \times 1$ would imply the assumption that local structure is important, which is inap-
propriate in our domain; thus, we do not need multiple mlpconv layers). Second, our pooling units are superficially similar to the pooling units used in convolutional networks. However, ours differ both in functional form and purpose: we use pooling as a way of sharing information between cells in the matrices that are processed through our network by taking maximums across entire rows or columns, while in computer vision, max-pooling units are used to produce invariance to small translations of the input image by taking maximums in a small local neighborhood.
Chapter 4

Representational Generality of Our Architecture

Our work aims to extend existing models in behavioral game theory via deep learning, not to propose an orthogonal approach. Thus, we must demonstrate that our representation is rich enough to capture models and features that have proven important in that literature. In this section we make explicit the connection between our model and popular models from the behavioral game theory literature by demonstrating how our architecture is able to express these models. At a high level, we express behavioral models using appropriately parameterized the action response layers and we express non-strategic features using the invariance preserving hidden layers with pooling units.

4.1 Behavioral Models

The four behavioral models we consider are quantal cognitive hierarchy (QCH) [30, 36] and quantal level-k (QLk) [30], cognitive hierarchy (CH) [3] and level-k (Lk) [8]. They differ in their behavioral assumptions, but they are similar in their mathematical descriptions. All involve some notion of a response (in the form of a strategy or distribution over one’s own actions) to beliefs about one’s opposition strategy (in the form of a distribution over one’s opposition’s actions).
Responses  We can divide these four models into two distinct classes: either they assume players best respond or quantally respond. Players best respond by selecting an action from the set of actions that maximizes their expected utility given their beliefs. Alternatively, they quantally respond by choosing actions with probability proportional to the action’s expected utility.

This is modeled formally as follows: let \( \bar{u}_i(s) = \sum_{j=1}^{n} u_{i,j} s_j \) denote a player’s expected utility, given beliefs, \( s \), about the opposition actions. A quantal response strategy is defined as

\[
s_i(s) = \frac{\exp(\lambda \bar{u}_i(s))}{\sum_{m=1}^{n} \exp(\lambda \bar{u}_i(s))}.
\]

Quantal response approaches best response as \( \lambda \to \infty \) in the sense that it defines a strategy where players uniformly randomize from the set of best responses\(^1\). Alternatively, if \( \lambda \to 0 \), players ignore their payoffs and uniformly randomize over their set of actions.

Beliefs  The four models can also be categorized based on how they define beliefs about one’s opposition, \( s_j \). All of the models rely on a notion of a “cognitive level” that differs among players. However, while Level-\( k \) models assume that a player at cognitive level \( k \) only has beliefs about level \( (k - 1) \) players, cognitive hierarchy models assume that players respond to beliefs about the full distribution of players having cognitive level less than their own.

We now show the connection between our neural network-based approach and these behavioral models. Recall that action response layer \( l \) is defined as

\[
\text{ar}_l^{(r)} = \text{softmax} \left( \lambda_l \left( \sum_{j=0}^{l-1} v_{l,j}^{(r)} \left( \sum_{i=1}^{k} w_{l,i}^{(r)} H_{L_i}^{(r)} \cdot \text{ar}_j^{(c)} \right) \right) \right), \quad \text{ar}_l^{(r)} \in \Delta^m, l \in \{1, \ldots, K\},
\]

and action response layer 0 is defined as a weighted sum of features, \( \text{ar}_0 = \sum_{l=1}^{K} w_l f_l \).

The behavioral models do not depend on transformed versions of the input

---

\(^1\)The claim that we can represent best response by letting parameters tend to infinity may appear dubious given that the models are optimized numerically. However because the \( \exp(x) \) function saturates quickly using floating point numbers, in practice \( \lambda \) only needs to be moderately large to output a best response.
matrices or behavioral features, so we let the parameters of the network be set such that
\[
\mathbf{ar}_l^{(r)} = \text{softmax} \left( \lambda_l \left( \sum_{j=0}^{l-1} v_{l,j}^{(r)} \mathbf{U}^{(r)} \cdot \mathbf{ar}_j^{(c)} \right) \right), \quad \mathbf{ar}_l^{(r)} \in \Delta^n, l \in \{1, \ldots, K\},
\]
and let \( \mathbf{ar}_{0,j} = \frac{1}{m} \) for all \( i \in \{1, \ldots, m\} \).

### 4.1.1 Level-\(k\)

The Level-\(k\) model associates every player in the population with a particular cognitive level corresponding to the number of steps of strategic reasoning they complete (bounded by some fixed maximum level), and assumes that level-\(k\) players best respond to the strategy played by level-(\(k-1\)) players and that level-0 players select actions uniformly at random. Each level also has some probability \( \varepsilon_k \) of making an “error” by selecting some action other than their best response.

We model this with the action response layers, by setting
\[
v_{l,i}^{(r)} = \begin{cases} 
1 - \varepsilon_i & \text{if } i = l - 1 \\
\varepsilon_i & \text{if } i = 0 \\
0 & \text{otherwise}
\end{cases}
\]
and letting \( \lambda_l \to \infty \) in order to simulate best response.

### 4.1.2 Cognitive Hierarchy

Cognitive Hierarchy is similar to level-\(k\) except it assumes a distribution over the levels a player may take, and assumes they best respond without error to the normalized distribution of players below them.

That is, there is some level distribution \( \mathbf{p} \) where \( p_i \) is the proportion of players in the population who behave according to a particular level and players respond to a normalized distribution \( \mathbf{p}_{[0,i-1]} \).

We model this with the action response layers, by setting
\[
v_{l,i}^{(r)} = \frac{p_i}{\sum_{j=0}^{i-1} p_j}
\]
and letting $\lambda_l \to \infty$ in order to simulate best response.

4.1.3 Quantal Level-$k$

Quantal Level-$k$ differs from the level-$k$ model described above by allowing players to quantally respond with each level having a different precision parameter $\lambda_l$. The parameters remain as described in Section 4.1.1 except that we use $\lambda_l$ instead of letting $\lambda \to \infty$.

4.1.4 Quantal Cognitive Hierarchy

Quantal Cognitive Hierarchy \[30, 36\] generalists cognitive hierarchy by allowing players to quantally respond and optimism the parameter $\lambda$. Similarly to the above, the parameters of our action response layers remain the same as in Section 4.1.2 except we use $\lambda_l = \lambda$ instead of letting $\lambda_l \to \infty$.

4.2 Game Theoretic Features

Wright and Leyton-Brown \[36\] showed the importance of explicitly modeling non-strategic level-0 players. They investigated a wide range of models and found that weighted linear combinations of nonstrategic features were most effective for improving predictive performance. In this section we argue that our model is sufficiently flexible to represent all the behavioral features used in their best performing models which allows us to generalize the quantal cognitive hierarchy with weighted linear features model presented in Wright and Leyton-Brown \[36\]. We make this claim formally below.

Claim: A network with two hidden layers, one hidden unit per layer, pooling units at every layer and rectified linear unit activation functions can represent each of the following normalized features

- $\min \max$ regret,
- $\min \min$ unfairness,
- $\max \min$ payoff,
• \textit{max max payoff},

• \textit{max max efficiency}.

The proof of this claim is laborious but straight forward because it simply amounts to selecting appropriate parameters to represent each features. It is given in Section A.1 in the Appendix. The key idea is that all of the above functions can be represented using a composition of max, min and absolute value operations. The pooling layers allow us to represent max and min functions, we can represent the absolute value function using a pair of rectified linear unit activations, and the network allows function composition by design.
Chapter 5

Experiments

5.1 Experimental Setup

We used a dataset that combined observations from 9 human-subject experimental studies conducted by behavioural economists in which subjects were paid to select actions in normal-form games. Their payment depended on the subject’s actions and the actions of their unseen opposition who chose an action simultaneously. The subjects were shown the payment for each outcome using a payoff matrix that lists each pair of actions and the respective player’s payments (see Figure 1.1 in Chapter 1 for an example). Each experiment presented subjects with between 8 and 20 different games and the number of subjects who selected each action is recorded. Obtained outcomes were only shown at the end of each experiment to prevent learning effects. The games range in size from $2 \times 2$ to $121 \times 121$; in the majority, players have three actions each.

Our model takes such payoff matrices as input and predicts the observed frequency with which each action was selected by the row player. We encode the payoffs of each normal form game as a pair of utility matrices corresponding to the respective players’ payoffs, normalised such that the standard deviation of the payoffs is approximately 1. For symmetric games we combine observations as though both players were the row player and for asymmetric games we treat observations of the column player’s choice of actions as though they had come from a game with a transposed payoff matrix, such that they become the row player. A few games ap-
Table 5.1: Our datasets. Each experiment had subjects play between 8 and 20 games for a total of 128 games, 113 of which were unique.

<table>
<thead>
<tr>
<th>Source</th>
<th>Games</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stahl and Wilson [30]</td>
<td>10</td>
<td>400</td>
</tr>
<tr>
<td>Stahl and Wilson [31]</td>
<td>12</td>
<td>576</td>
</tr>
<tr>
<td>Costa-Gomes et al. [7]</td>
<td>18</td>
<td>1296</td>
</tr>
<tr>
<td>Cooper and Van Huyck [6]</td>
<td>8</td>
<td>2992</td>
</tr>
<tr>
<td>Rogers et al. [23]</td>
<td>17</td>
<td>1210</td>
</tr>
<tr>
<td>Haruvy et al. [15]</td>
<td>15</td>
<td>869</td>
</tr>
<tr>
<td>Haruvy and Stahl [14]</td>
<td>20</td>
<td>2940</td>
</tr>
<tr>
<td>Stahl and Haruvy [29]</td>
<td>18</td>
<td>1288</td>
</tr>
<tr>
<td>All9</td>
<td>113 unique</td>
<td>12071</td>
</tr>
</tbody>
</table>

pear in multiple experiments; we combined their observed frequencies into a single common game.

We evaluate performance of the models using 10-fold cross-validation. We randomly partitioned our 113 unique games into 10 folds containing between 11 and 15 games each. Our experimental results examine both the mean and variance across 10 different such 10-fold cross-validations, each time re-randomising the assignment of games into folds (requiring each model to be retrained 100 times).

We are interested in the model’s ability to predict the distribution over the row player’s action, rather than just its accuracy in predicting the most likely action. As a result, we fit models to maximise the likelihood of training data \( P(D|\theta) \) (where \( \theta \) are the parameters of the model and \( D \) is our dataset) and evaluate them in terms of negative log-likelihood on the test set.

5.2 Training Details

All the models presented in the experimental section were optimized using Adam [16] with an initial learning rate of 0.0002, \( \beta_1 = 0.9, \beta_2 = 0.999 \) and \( \epsilon = 10^{-8} \). The models were all regularised using Dropout with probability 0.2 and \( L_1 \) regularization with parameter = 0.01. They were all trained until there was no training set improvement up to a maximum of 25,000 epochs and the parameters from the
iteration with the best training set performance was returned.

Our architecture imposes simplex constraints on the mixture weight parameters. Fortunately, simplex constraints fall within the class of simple constraints that can be efficiently optimized using the projected gradient algorithm first proposed by Goldstein [12]. The projected gradient algorithm modifies standard gradient decent by projecting the relevant parameters onto the constraint set after each gradient update.\(^1\)

### 5.3 Experimental Results

Figure 5.1 shows a performance comparison between a model built using our deep learning architecture with only a single action response layer (i.e. no iterative reasoning; details below) and the previous state of the art, quantal cognitive hierarchy (QCH) with hand-crafted features (shown as a blue line); for reference we also include the best feature-free model, QCH with a uniform model of level-0 behaviour (shown as a pink line). We refer to an instantiation of our model with \(L\) hidden layers and \(K\) action response layers as an \(N+K\) layer network. All instantiations of our model with 3 or more layers significantly improved on both alternatives and thus represents a new state of the art. Notably, the magnitude of the improvement was considerably larger than that of adding hand-crafted features to the original QCH model.

Figure 5.1 considers the effect of varying the number of hidden units and layers on performance using a single action response layer. Perhaps unsurprisingly, we found that a two layer network with only a single hidden layer of 50 units performed poorly on both training and test data. Adding a second hidden layer resulted in test set performance that improved on the previous state of the art. For these three layer networks (denoted (20, 20), (50, 50) and (100, 100)), performance improved with more units per layer, but there were diminishing returns to increasing the number of units per layer beyond 50. The four-layer networks (denoted (50, 50, 50) and (100, 100, 100)) offered further improvements in training set performance but test set performance diminished as the networks were able to overfit.

\(^1\)We adapted Schmidt et al. [25]’s implementation of the simplex projection operation to optimise our model.
Figure 5.1: Negative Log Likelihood Performance. The error bars represent 95% confidence intervals across 10 rounds of 10-fold cross-validation. We compare various models built using our architecture to a quantal cognitive hierarchy (QCH) model with uniform level-0 behavior (pink line) and QCH with a weighted linear model of level-0 behavior (blue line).

To test the effect of pooling units on performance, in Figure 5.2 we first removed the pooling units from two of the network configurations, keeping the rest of the hyperparameters unchanged. The models that did not use pooling layers underfit on the training data and performed very poorly on the test set. While we were able to improve their performance by turning off dropout, these unregularised networks didn’t match the training set performance of the corresponding network configurations that had pooling units. The test set performance for all the networks we trained without pooling units remained significantly worse than our best performing networks that used pooling units. Thus, our final network contained two layers of 50 hidden units and pooling units and used dropout.
Figure 5.2: Performance comparison with and without pooling units. All four models were fit with the same hyperparameters using dropout unless otherwise stated, with the only difference being the number of layers and hidden units and whether or not the models used pooling units.

Our next set of experiments committed to this configuration for feature layers and investigated configurations of action-response layers, varying their number between one and four (i.e., from no iterative reasoning up to three levels of iterative reasoning; see Figure 5.3). The networks with more than one action-response layer showed signs of overfitting: performance on the training set improved steadily as we added AR layers but test set performance suffered. Thus, our final network used only one action-response layer. We nevertheless remain committed to an architecture that can capture iterative strategic reasoning; we intend to investigate more effective methods of regularising the parameters of action-response layers in future work.
Figure 5.3: Performance comparison as we vary the number of action response layers. The AR layers appear to result in overfitting in the network as increasing the number of AR layers reduces the average training set loss while increasing the average test set loss.

5.4 Regular Neural Network Performance

Figure 5.4 compares the performance of our architecture with that of a regular feed-forward neural network, with and without data augmentation, and the previous state-of-the-art model on this dataset. It shows that the feed-forward network dramatically overfitted the data without data augmentation. Data augmentation improved test set performance, but it was still unable to match state of the art performance. A three layer instantiation of our model (two layers of 50 hidden units and a single AR layer) matched the previous state of the art but failed to improve upon it. We suspect that this may be because the subset of the data that contains only $3 \times 3$ games is too small to take advantage of the flexibility of our model.
Figure 5.4: Performance comparison on $3 \times 3$ games of a feed-forward neural network (FFNet), a feed-forward neural network with data augmentation at every epoch (FFNet (Permuted)), our architecture with two hidden layers of 50 units each (labeled (50, 50)), and quantal cognitive hierarchy with four hand-crafted features (QCH Linear4).
Chapter 6

Discussion and Conclusions

To design systems that efficiently interact with human players, we need an accurate model of boundedly rational behavior. This thesis presented an architecture for learning such models that both significantly improves upon state-of-the-art performance without needing hand-tuned features developed by domain experts. Interestingly, while the full architecture can include action response layers to explicitly incorporate the iterative reasoning process modeled by Level-k-style models, our best-performing model did not need them to achieve set a new performance benchmark. This indicates that the model is performing the mapping from payoffs to distributions over actions in a manner that is substantially different from previous successful models.

The model presented in this work can represent a large class of nonstrategic game-theoretic features but it is not universal. For example, we can’t model behavior where players are attracted to particular actions that are salient because of their location in the payoff matrix or if they avoid actions because they are dominated by some other action. We would like to remove this limitation in future work. Some natural future directions, are to extend our architecture beyond two-player, unrepeated games to games with more than two players, as well as to richer interaction environments, such as games in which the same players interact repeatedly and games of imperfect information.
Bibliography


Appendix A

Supporting Materials

A.1 Representing behavioural features

Claim: A network with two hidden layers, one hidden unit per layer, pooling units at every layer and rectified linear unit activation functions can represent each of the following normalised features,

- min max regret,
- min min unfairness,
- max min payoff,
- max max payoff
- max max efficiency.

Proof By expanding the sums from the definition of the network, we see the first hidden layer has the following functional form:

\[ H^{(1)} = \text{relu}(w_{1,r}U^{(r)} + w_{1,c}U^{(c)} + w_{1,rc}U^{(r)}_{c} + w_{1,cc}U^{(c)}_{r} + w_{1,cr}U^{(r)}_{c} + w_{1,cc}U^{(c)}_{r} + b_{1,1}). \]

where \( U^{(r)} \) is the row player’s payoff matrix and \( U^{(c)} \) is the row player’s payoff matrix aggregated using the column-preserving pooling unit where we use the max
function to perform the aggregation. Similarly, the second hidden layer can be written as,

\[ H^{(2,1)} = \text{relu}(w_{2,1}H^{(1,1)} + w_{2,c}H_{c}^{(1,1)} + w_{2,r}H_{r}^{(1,1)} + b_{2,1}). \]

We denote \( H^{(1,1)} \) as the output of the first hidden layer and \( H_{c}^{(1,1)} \) and \( H_{r}^{(1,1)} \) are its respective pooled outputs.

Game theoretic features can be interpreted as outputting a strategy (a distribution over a player’s actions) given a description over the game. We express features in a style similar to [36] by outputting a vector \( f \) such that \( f_i \approx 0 \) for all \( f_i \in f \) if action \( i \) does not correspond to the target feature, and \( f_i \approx l \) where \( l \) is the number of actions that correspond to the target feature (with \( l = 1 \) if the actions uniquely satisfies the feature; Wright and Leyton-Brown [36] instead used a binary encoding, but that does not fit naturally into our framework). We have approximate equality, \( \approx \), because we construct the features using a softmax function and hence our output approaches \( f_i = 0 \) or \( 1 \) as our parameters \( \to \infty \). Because features are all constructed from a sparse subset of the parameters, we limit notational complexity by letting \( w_{i,j} = 0 \) and \( b_{i,j} = 0 \) for all \( i, j \in 1, 2, r, c \) unless stated otherwise.

**A.1.1 Max Max Payoff**

**Required:** 

\[ f^{\text{maxmax}}(i) = \begin{cases} 
\frac{1}{l} & \text{if } i \in \arg\max_{i \in \{1, \ldots, m\}} \max_{j \in \{1, \ldots, n\}} u_{i,j} \\
0 & \text{otherwise}
\end{cases} \quad u_{i,j} \in U^{(r)} \]

Let \( w_{1,r} = 1, w_{2,r} = c \) where \( c \) is some large positive constant and \( b_{1,1} = b \) where is some scalar \( b \geq \min_{i,j} U^{(r)}_{i,j} \) and all other parameters \( w_{i,j}, b_{i,j} = 0 \). Then \( H^{(1,1)} \) reduces to,

\[ H^{(1,1)} = \text{relu}(U^{(r)} + b) = U^{(r)} + b \quad \text{since } U^{(r)} + b \geq 0 \text{ by definition of } b \]

\[ H^{(2,1)} = \text{relu}(cH_{c}^{(1,1)}) \Rightarrow h_{j,k} = c(\max_{k} u_{j,k} + b) \quad \forall u_{j,k} \in U^{(r)}, h_{j,k} \in H^{(2,1)} \]

That is, all the elements in each row of \( H^{(2,1)} \) equal an positive affine transformation
of the maximum element from the corresponding row in $U^{(r)}$.

$$f_i^{(1)} = \text{softmax}(\sum_{k=1}^{n} h_{j,k}) = \text{softmax} \left( \sum_{k=1}^{n} c(\max u_{j,k} + b) \right) = \text{softmax} \left( nc(\max u_{j,k} + b) \right)$$

Therefore, as $c \to \infty$, $f_i^{(1)} \to f^{\text{maxmax}}(i)$ as required.

### A.1.2 Max Min Payoff

**Required:**

$$f^{\text{maxmin}}(i) = \begin{cases} 
\frac{1}{l} & \text{if } i \in \arg \max_{i \in \{1, \ldots, m\}} \min_{j \in \{1, \ldots, n\}} u_{i,j} \quad u_{i,j} \in U^{(r)} \\
0 & \text{otherwise} 
\end{cases}$$

Max Min Payoff is derived similarly to Max Max except with $w_{1,r} = -1$, and $b_{1,1} = b$ where $b \geq \max_{i,j} U^{(r)}_{i,j}$; we keep $w_{2,r} = c$ as some large positive constant.

Then $H^{(1,1)}$ reduces to,

$$H^{(1,1)} = \text{relu}(-U^{(r)} + b) = -U^{(r)} + b \quad \text{since } -U^{(r)} + b \geq 0 \text{ by definition of } b$$

$$H^{(2,1)} = \text{relu}(cH^{(1,1)}_{r}) \Rightarrow h_{j,k} = c(\max(-u_{j,k} + b)) = c(\min u_{j,k} + b) \quad \forall u_{j,k} \in U^{(r)}, h_{j,k} \in H^{(2,1)}$$

Since $\max x_i - x_i + b = \min_i x_i + b$. Thus,

$$f_i^{(1)} = \text{softmax}(\sum_{k=1}^{n} h_{j,k}) = \text{softmax} \left( nc(\min u_{j,k} + b) \right)$$

Therefore, as $c \to \infty$, $f_i^{(1)} \to f^{\text{maxmin}}(i)$ as required.

### A.1.3 Max Max Efficiency

**Required:**

$$f^{\text{max max efficiency}}(i) = \begin{cases} 
\frac{1}{l} & \text{if } i \in \arg \max_{i \in \{1, \ldots, m\}} \max_{j \in \{1, \ldots, n\}} u_{i,j}^{(c)} + u_{i,j}^{(r)} \\
0 & \text{otherwise} 
\end{cases}$$

Max Max Efficiency follow from the derivation of Max Max except with $w_{1,r} = 1, w_{1,c} = 1, w_{2,r} = c$ and $b_{1,1} = b$ where $b \geq \min_{i,j}(U^{(r)}_{i,j} + U^{(r)}_{i,j})$. 

41
Following the same steps we get,

\[ f_i^{(1)} = \text{softmax}\left( \sum_{k=1}^{n} h_{j,k} \right) = \text{softmax}\left( \sum_{k=1}^{n} c \left( \max_k \left( u^{(r)} + u^{(c)} \right)_{j,k} + b \right) \right) \]

\[ = \text{softmax}\left( nc \left( \max_k \left( u^{(r)} + u^{(c)} \right)_{j,k} + b \right) \right) \]

\[ = f^{\text{max max efficiency}}(i) \quad \text{as } c \to \infty \]

A.1.4 Minimax Regret

Regret is defined as

\[ r(i,j) = \max_i u_{i,j} - u_{i,j} \quad u_{i,j} \in U^{(r)} \]

**Required:** \[ f^{\text{minimax regret}}(i) = \begin{cases} \frac{1}{l} & \text{if } i \in \arg \min_{i \in \{1, \ldots, m\}} \max_j r(i,j) \\ 0 & \text{otherwise} \end{cases} \]

Let \( w_{1,rc} = 1 \), \( w_{1,r} = -1 \), and \( b_{1,1} = 0 \); we keep \( w_{2,r} = c \) as some large positive constant.

Then \( H^{(1,1)} \) reduces to,

\[ H^{(1,1)} = \text{relu}(U^{(r)}_c - U^{(r)}) = U^{(r)}_c - U^{(r)} \quad \text{since } U^{(r)}_c \geq U^{(r)} \text{ by definition of } U^{(r)}_c \]

\[ H^{(2,1)} = \text{relu}(cH^{(1,1)}_r) \Rightarrow h_{j,k} = c \left( \max_j \left( \max_k u_{j,k} - u_{j,k} \right) \right) \quad \forall u_{j,k} \in U^{(r)}, h_{j,k} \in H^{(2,1)} \]

Thus,

\[ f_i^{(1)} = \text{softmax}\left( \sum_{k=1}^{n} h_{j,k} \right) = \text{softmax}\left( nc \left( \max_j \left( \max_k u_{j,k} - u_{j,k} \right) \right) \right) \]

Therefore, as \( c \to \infty \), \( f_i^{(1)} \to f^{\text{minimax regret}}(i) \) as required.

A.1.5 Min Min Unfairness

**Required:** \[ f^{\text{min min unfairness}}(i) = \begin{cases} \frac{1}{l} & \text{if } i \in \arg \max_{i \in \{1, \ldots, m\}} \min_j \left| u^{(r)}_{i,j} - u^{(c)}_{i,j} \right| \\ 0 & \text{otherwise} \end{cases} \]

To represent Min Min Unfairness, we add an additional hidden unit in the first layer.
Let \( H^{(1,2)} \) be defined in the same manner as \( H^{(1,1)} \).

For \( H^{(1,1)} \), we let \( w_{1,r}^{1} = 1, w_{1,c}^{1} = -1 \) and \( b_{1,1} = 0 \) such that,

\[
H^{(1,1)} = \text{relu}(U^{(r)} - U^{(c)}) = \max(U^{(r)} - U^{(c)}, 0)
\]

where the max is applied element-wise.

For \( H^{(1,2)} \), we let \( w_{1,r}^{1} = -1, w_{1,c}^{1} = -1 \) and \( b_{1,1} = 0 \) such that,

\[
H^{(1,2)} = \text{relu}(U^{(c)} - U^{(r)}) = \max(U^{(c)} - U^{(r)}, 0)
\]

Now, notice that if \( w_{2,1}^{1} = 1 \) and \( w_{2,2}^{1} = 1 \),

\[
H^{(2,1)} = H^{(1,1)} + H^{(1,2)} = \max(U^{(r)} - U^{(c)}, 0) + \max(U^{(c)} - U^{(r)}, 0) = |U^{(r)} - U^{(c)}|
\]

Which gives us a measure of “unfairness” as the absolute difference between the two payoffs.

We can therefore simulate \( f_{i}^{\min \text{ unfairness}} \) by letting \( w_{2,1}^{1} = -1 \) and \( w_{2,2}^{1} = -1 \), and using the output of \( H^{(2,1)} \) (which gives us min unfairness), then constructing \( f_{i} \) by letting \( c \to \infty \).