GameNetViz Final Report

Neil Newman* Jason Hartford†

December 19, 2015

Abstract

In complex machine learning models, it is often time-consuming and difficult for practitioners to see the intermediate computational sets that lead to the model's solution. We present GameNetViz as system for visualising the intermediate computation sets in a complex behavioural game theory model. It leverages off a particularly useful property of the domain: the model repeatedly transforms small matrices and small vectors; while the model itself is highly parameterized and complicated, the intermediate outputs are low dimensional and simple. This allows direct visual encoding that is effective at showing all intermediate transformations in a single overview view, with further details available via tool-tips and linked-highlighting.

1 Introduction

Behavioural game theory aims to predict the behaviour of people as they interact strategically. Researchers in the field typically have two goals: explaining how observed behaviour deviates from perfect rational behaviour, and maximising predictive accuracy.

With both of these goals in mind, we turn to modelling human strategic behaviour through Deep Learning. The recent success of Deep Learning in prediction tasks [4] such as image recognition [3] and natural language processing [11] has shown that for many problems, one can maximise a model's predictive accuracy by optimising flexible models composed of multiple processing layers to observed data. In certain domains, such as vision, the early layers of the model correspond to features that can be readily understood. However, adding flexibility to a model makes it both more difficult to optimise and more difficult to interpret, which slows research progress as it becomes difficult to detect modeling failures.

Through this project, we show that careful use of visualisation provides a useful tool to "see under the hood" of a complex model to see the mechanisms

*newmanne@cs.ubc.ca
†jasonhar@cs.ubc.ca

with which it is achieving its performance and develop hypotheses as to the causes of poor performance.

Our solution is *GameNetViz*, a tool that interacts directly with the behavioural model in order visualise the intermediate steps of computation. By exploiting the model's relative low-dimensionality, we are able to display a full overview of these steps on a per-data point basis that makes it easy to explore data points on which the model was more or less successful.

We proceed as follows: Section 2 surveys other attempts to visualise complex models in the literature, Section 3 describes the data and tasks, Section 4 presents our visualisation solution, Section 5 describes a typical interaction with the tool, Section 6 describes the tools we used to implement the visualisation, Section 7 discusses the strengths and limitations of the approach and Section 8 concludes.

2 Related Work

2.1 Model Visualisation

The closest prior work that we are aware of is [7], which develops an interactive system for validation of relatively complex regression models that allows researchers to understand the model's behaviour under different optimisation regimes. They demonstrate the usefulness of allowing engineers to visualise slices of the target function in order to get a feeling for the model's behaviour around a particular point in space.

The deep learning literature has also attempted to visualise components of their models in order to better understand how they work. In [8], the authors build functions that simulate the loss surfaces of challenging non-convex optimisation problems. To support this work, they use random projections to visualise slices of their target function.

In [5], the authors present a framework for interactively building regression models. Their system is centered around two small multiples overviews that show both single and pairwise relationships between features and the regression target value, ranked by a variety of relevance metrics. This approach is particularly effective for regression models because users can see the difference between the target and predicted value for each data point on each feature axis. Unfortunately, in our context, the model outputs a probability distribution and hence can't be directly represented by a single target value. Because regression models can be fit quickly, their system also allows interactive changes to the regression model. They use linked 3d views effectively for validation of their model.

[12] is a tool to systematically explore a multi-dimensional parameter space that affects the quality of image segmentation algorithms. Unlike regular pairwise small-multiples which typically only show the lower triangle of the pairwise matrix, Tuner uses both triangles in order to display two different optimisation

objectives simultaneously.

[9] describes *visual parameter space analysis*: systems where the inputs contain parameters that can be tuned that will affect the quality of the output. The paper distinguishes between different types of input: *control parameters*, which are tuned, and *environmental parameters* which come from measured data. The paper also codifies analysis tasks: using this paper's terminology, we have an *optimization* task, since we are interested in finding the best parameters given a loss function, as well as a *sensitivity* task, since we are interested in studying the impact on outputs of varying the model's input parameters. However, unlike this paper, our we cannot view the model as a black box, since we want to understand what it is that the black box is doing.

In contrast, [13] provides a viz tool with the goal of understanding the relative importance of input and hidden units in neural networks. The importance of a weight (a metric derived from propagating the weight throughout the network) is encoded in the width of the connection, and the importance of an input unit is encoded by its size. The authors were able to use these visualizations to identify what the important features were in a spam classifier, and to remove unimportant nodes from the hidden layers (leading to a more compact network with similar accuracy). However, the examples in the paper only have a single hidden layer, and it is unclear how these techniques would generalize to networks with multiple hidden layers.

2.2 Feature discovery

[14] visualises the layers in a convolutional network that is used for image classification, using a derived data technique that allows them to infer the input pixels that resulted in particular outputs in intermediate layers in the network. Unfortunately, this technique relies on the visual structure encoded in the parameters of the convolutional network for be useful, and thus doesn't generalise to our domain. This limitation motivates our interest in comparing learnt features to hand

3 Data and Task Abstractions

3.1 Data Abstraction

There are two sources of data that we have to consider in this project: the raw data describing the games and observed behaviour, and the parameters of a machine learning model which attempts to predict behaviour given a description of a game.

A game is described by a *payoff matrix*: $n \times m$ tables where n is the number of decisions available to the first player and m is the number of decisions available to a second player. Each (i, j) cell in the table contains a tuple of integer values that describes the payoff that each player will get if the first player chooses action i and the second player chooses action j. The raw data comes from behavioural

economics studies involving real subjects playing a variety of one-shot games with each other. One-shot means that the games entirely consist of each user making a single decision from a given set of decisions, without observing any previous play from their opponent.

There are 128 unique payoff matrices in the dataset, and $12,071$ observations of players playing these games (consisting of the choices that the players made). Therefore, there are 128 distributions over the actions for each player. Payoff matrices in the dataset range in size from 2×2 to 5×5, with the majority being 3×3 games.

The machine learning model can be viewed as a composition of functions that map the payoff matrices to a vector of probability distributions describing the predicted frequency with which a player will select a particular action. The function composition can be described by a tree where the root nodes are the function inputs, each internal node indicates the application of a function to its parent's output, and the functions are parametrised by the weights of the in-edges. Domain experts are primarily interested in the parameters of the function (the edges in the graph), and the intermediate function outputs. Of particular importance is the notion of *learnt features* which are intermediate predictions of probabilities that players select actions that the model uses to make its final prediction.

A full summary of the domain specific terms is given in Table 1.

3.2 Task Abstraction

We aim to solve two related tasks using this visualisation:

- Show practitioners how the model constructs its output from intermediate computation steps.
- Compare the model's predictions on individual data points to what actually occurred in experiments.

We describe each task in turn.

3.2.1 Computation Steps

The *GameNet* model takes a matrix of payoffs as inputs and performs a number of steps of computation to produce its output. Each of these computational steps can be thought of as transformations of either the input payoff matrix or an earlier transformation's output.

Each transformation occurs in a *unit* and the *units* fall into four distinct classes, each requiring different visual encoding.

- *Hidden units* are matrix to matrix transformations. They output matrices formed through a weighted sum of either the payoff matrix or previous hidden units followed by a non-linearity. These are the most challenging objects to visualise because the magnitude of their outputs are unconstrained and each output may contain up to 5×5 values.

Domain	Mathematical Abstraction	Visual Abstraction
The object of interest is the *model* which outputs a probability distribution over *actions* for the second player in each *game* in the dataset.	The *model* is a function $F_\theta : \mathbb{R}^{n\times m} \cup \mathbb{R}^{n\times m} \to \mathbb{S}^n$ that maps a *game* defined by a *payoff matrix* to the probability simplex. The function is parametrised by a set of *parameters* θ. The model $F = f_1 \circ f_2 \circ \dots f_l$ is a composition of multiple functions f_i which is best visualised as a network where nodes represent functions and edges show the mapping between functions.	The model takes a multidimensional table of numbers which are repeatedly transformed by the functions $f_1, \dots, f_n$ in proportion to the parameters. We visualise this as a series of intermediate states.
A *game* is a strategic interaction between one or more players who may each choose from a set of *actions*. This project focuses on a model that applies to two-player games where each player has between 2 and 5 *actions*	Formally, a normal-form game is a tuple (N, A, u) where N is a set of players, A is the set of actions available to each player and $u_i : A \to u_i$ is a mapping from actions to payoffs. For our purposes it suffices to think of a game as being an object defined by a *payoff matrix*.	A *game* is represented as a *payoff matrix* which is a multidimensional table indexed by a pair of actions, where the value is a tuple of numbers corresponding to the payoffs
Feature is a distribution over the *actions* available to a player indicating the probability that the player may choose that action.	Each feature f_i is a vector in $\mathbb{S}^n$, the n-dimensional simplex, where n is the number of actions available to the player.	A vector of positive real numbers that sum to 1.

Table 1: Summary of Domain-Specific Terms

- *Feature units* are matrix to vector transformations where the vectors all sum to one, and each element of the vector corresponds to the probability the feature assigns to a particular *action*.
- *Action Response units* are vector to vector transformations that are arranged in levels and by player. An Action Response layer for a particular player at level k adds a weighted sum of the feature vectors to a weighted sum of the opposite player's Action Response $k-1$ preceding unit's outputs before applying a non-linear transformation that outputs a vector

that sum to one.

- *Output units* are vector to vector transformations which are composed of the weighted sum of player 1's Action Response units. The output vector sums to one.

3.2.2 Performance Comparison

For each game in the data set we have a model prediction and the result of what occurred in the experiment. A typical task for practitioners is to compare the performance in the model to what actually occurred in order attempt to understand whether there is anything systematic in the games on which the model is performing poorly which may suggest deficiencies in the model.

To support this, the visualisation needs to provide a mechanism for comparing predicted to actual performance, and a way of evaluating the characteristics of a game.

4 Solution

This section will describe GameNetViz. The overall system can be seen in Figure 1. A major feature of our design is that computation flows from left to right in the visualization. We begin with the inputs to the model, which are the payoff matrices on the far left, and these are followed by the hidden layers, feature layers, action response layers, and finally end with the the output layer on the far right. Our visualization directly mirrors the layout of the model's internals, and as a result, any inputs to a computational unit are always to its left.

4.1 Payoff Matrix

The input to the model is a payoff matrix. We encode the input in two ways which the user can switch between at any time that we refer to as Number and Blob views. Our two designs are shown in Figure 2. The Number view is the traditional game theory representation of the payoff matrix: each cell lists the payoff for each player if the corresponding actions are chosen. The weakness of this view is that the user may have to read $m \times n \times 2$ numbers in order to understand the game. Therefore, we also provide the Blob view: in this view, each cell contains two shaded circles whose areas are proportional to the payoff that the user will receive. This view provides several advantages: Firstly, it is very easy to see when the payoffs for each player are about the same or wildly different: if a cell contains payoffs of very different sizes for each player, it will be immediately obvious from this view. Secondly, it is very easy to spot zero payoffs, as no circle is drawn for a player with a zero payoff. The payoffs are normalized relative to the highest payoff value in the matrix, so that the largest value in a game will always take up the same amount of area. It would have been possible to normalize against the largest overall payoff in any of the games, but the relative scales of games in the dataset is very different (some games have

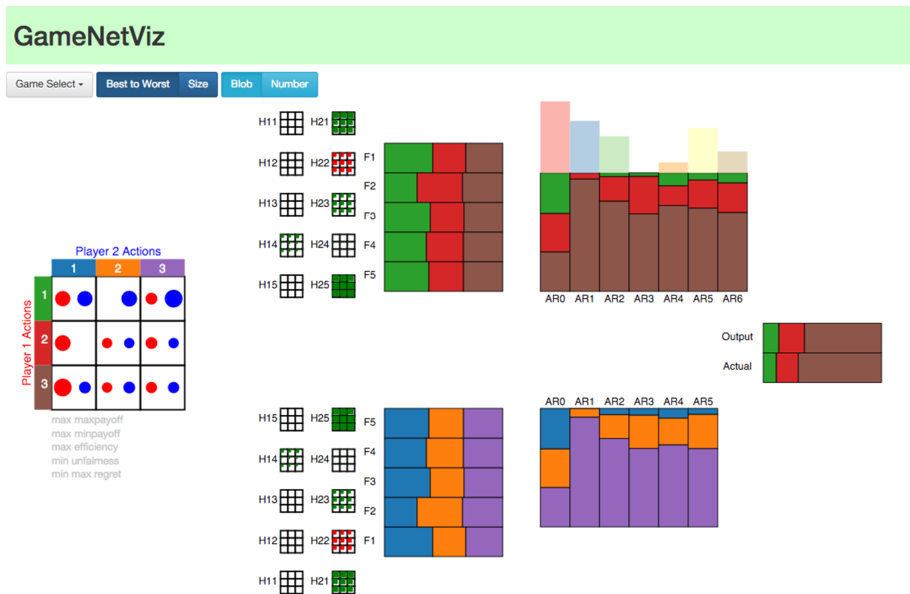

Figure 1: The full GameNetViz interface

all small payoffs, some games have all large payoffs, some games have both) and this type of normalization would have made this view useless for games where all of the payoffs are small. While the Blob view is great for a quick summary, it is almost impossible to distinguish between a payoff of 300 a payoff of 301, and it therefore it cannot replace the Number view (for a game where the only 2 possible payoffs are 300 and 301, this could matter).

4.2 Hidden Layers

The payoff matrix undergoes a series of transformations through the hidden layers. There are a number of hidden layers (the number is a choice of the model) and each layer is composed of a number of hidden units (also a choice of the model). Each hidden unit outputs a matrix with the same dimensions as the payoff matrix, with the important distinction that this matrix is single valued (instead of a pair of numbers). We encode these outputs in our visualization, which can be seen in 3. For each hidden unit, we draw a grid of the same dimensions of the output matrix. In each grid cell, we shade an area proportional to the magnitude of the output. The colour of the shaded area is green if the value is positive, and red if the value is negative. Normalization is performed relative to the largest value in any hidden unit for the particular game being displayed.

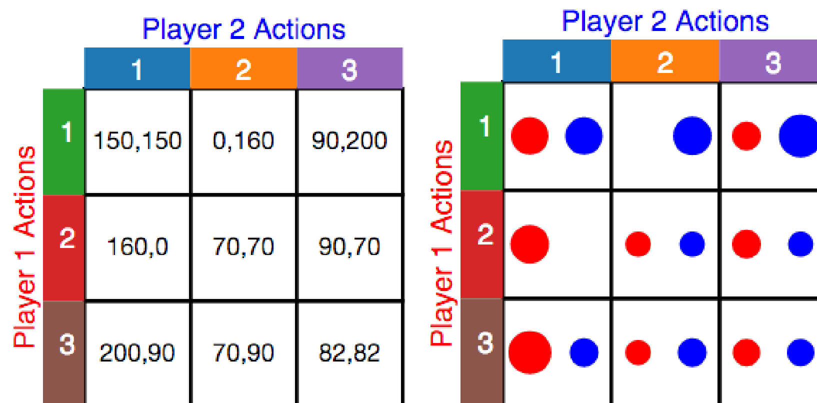

Figure 2: The Payoff matrices showing the traditional numeric representation (left) and the blob view (right)

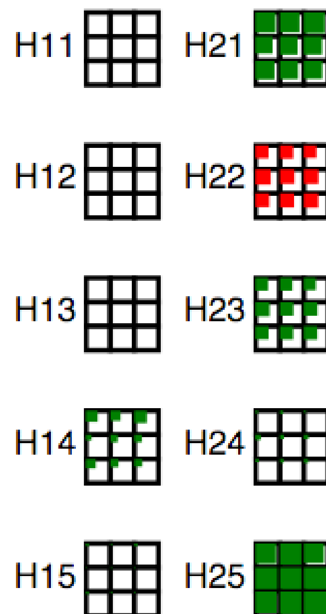

Figure 3: The hidden layers. Shaded block area represents the magnitude of the number in the cell in the matrix. Positive numbers are shown in green and negative numbers in red.

4.3 Features

The feature layer of the model is composed of the final hidden layer. The number of features for each player is specified by the model. Each feature outputs a probability distribution over a player's actions. We encode these distributions as stacked bar charts: one for each feature, piled on top of one another as shown in Figure 4. The order of the actions in each of the stacked bar charts is constant. The user can see an overview of what the features as a whole are predicting by by examining the relative amounts of area devoted to each colour in the piled up figure. To make comparisons across features, the user can interact with the stacked bar charts by clicking on them. The stacked bar charts will separate into grouped bar charts (where the grouping is by action), making cross feature comparisons for a particular action easy. The stacked bar charts expand to the right, and the rest of the visualization flies off screen to the right to make room for the group bar charts; this animation ensures the user does not get lost. Clicking on the grouped bar charts reverts the visualization to its former state.

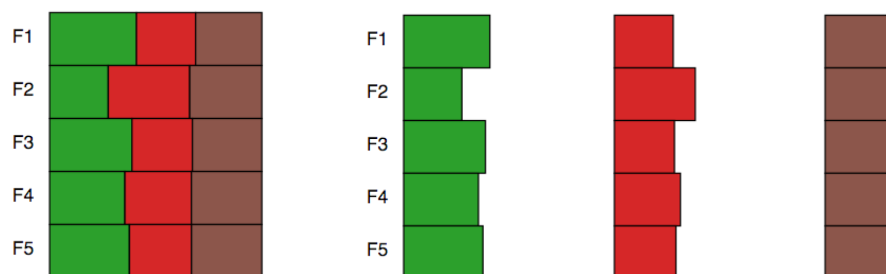

Figure 4: Each feature outputs a different probability distribution of playing each action represented as stacked bar charts. Clicking on the stacked bar charts splits them into grouped bar charts, so that you can compare an action's distribution across different features.

4.4 Action Response Layers

The action response layers, similarly to the feature layers, output probability distributions over actions. Unlike the feature layers, action response layers are composed of previous action response layers. Therefore, in order to maintain our left to right metaphor of computation, we needed to draw each action response layer's stacked bar chart vertically in order for the entire visualization to be viewable on screen at once (see Figure 5). When the user hovers over an action response layer, a tooltip pop up (see Figure 6). An action response player is composed of contributions from the feature layers, as well as from previous action response layers. These contributions are summed up and passed through a non-linearity. The tooltip shows the relative contribution from the feature layers and the action response layers, as well as their sum. The tooltip is to the

left of the action response layer, so that the output (the sum passed through the non-linearity) can also be seen. The non-linearity is controlled by a sharpness parameter (unique to each action response layer), which is also displayed at the bottom of the tooltip. Above the action response layers, we include a bar chart representing the level distribution. The output is composed of a weighted sum of each of the action response layers, and the weights are given by this level distribution.

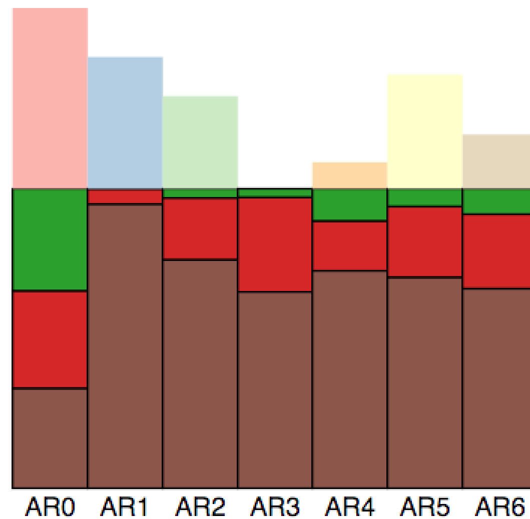

Figure 5: The Action Response layers output probability distributions over actions represented as stacked bar charts. Above the AR layers we show the level distribution indicating the layer's contribution to the final output

4.5 Output

The final prediction of the model is a probability distribution over the actions for the second player. This probability distribution is juxtaposed with the actual observed distribution of play from experiments, as shown in Figure 7. Again, we used the stacked bar chart encoding.

4.6 Hover

In order to determine what parts of the model influence a particular computational unit, a user can hover over a unit which will cause edges to be rendered from that unit to all of its inputs. An example of this hover action can be seen in Figure 8.

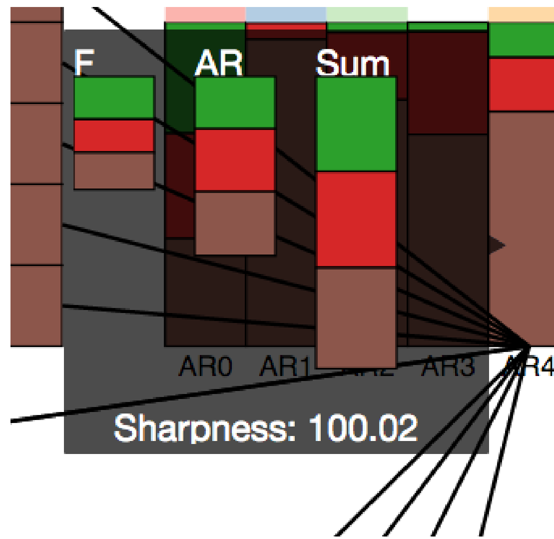

Figure 6: Hovering over an AR unit produces a tool-tip showing a breakdown of how it is composed, before the non-linearity is applied. The sharpness parameter of the non-linearity function for that AR layer is also displayed

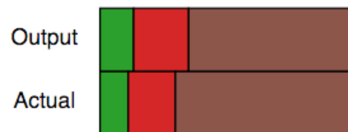

Figure 7: The output compares the actual output to the predicted output

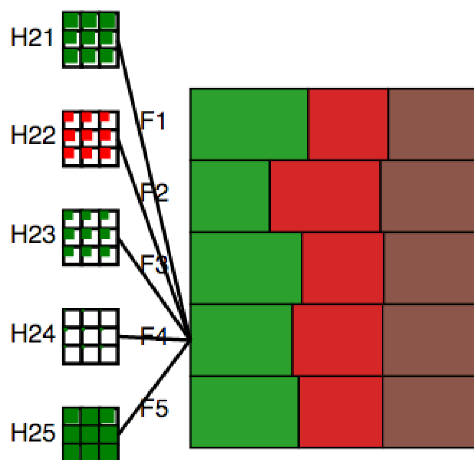

Figure 8: Edges give a quick visual indication of the preceding nodes on which the unit depends

4.7 Game Selection

The model's predictions are all on a per-game basis. The user can change the game that is being displayed by selecting from a dropdown menu located at the top left of the screen as show in in Figure 9. A new selection causes the entire display to be redrawn. There are two sorting methods for the games: by game size (e.g. by games with 4 actions for player 2 and 2 actions for player 1) or by a derived metric of the model's performance, allowing the user to easily see where the model is doing poorly.

4.8 Hand Crafted Features

As mentioned previously, there are several known features that are good indicators of how people will interact. For example, the max max payoff feature corresponds to the maximum possible achievable payoff for a given player. Below each payoff matrix, we list five of these features in greyed out text. If the user hovers over one of these feature labels, the feature text becomes more prominent and the rows in the matrix corresponding to the actions associated with that feature on that game are highlighted, as can be seen in Figure 10. There are some limitations to our current approach: we only show the features for the first player, though it would also be interesting to highlight columns showing the features for the second player. We also highlight entire rows, when it would be preferable to also highlight the individual outcomes that are triggering the feature. For example, the outcome that provides the max max payoff

Figure 9: Choosing a game from the selector will render data for that game. Games can be ordered by size, or by a derived difference between the model's prediction and observed play

may only be achievable if the other player selects a particular action, and we should highlight this fact. These improvements were not made purely due to time constraints.

5 Scenario of use

The user has trained his model and is now interested in understanding where it performs well and where it performs poorly. He loads the model into GameNetViz and selects a game from his data set using the game selector (Figure 9).

He begins by comparing the actual to predicted output. At a glance he can see how similar the two are from the grouped bar chart.

For games on which the model is performing well, he may be interested in which components of the model lead to its performance. By examining the level distribution above the AR layers (Figure 5) two observations are immediately obvious. AR3 is not contributing at all to the solution and much of the performance is derived from AR0. By hovering over individual AR layers a tool tip pops up to show the contribution of Features and AR layers to the AR-layer's performance (see Figure 6).

Similarly, he examines the output of the feature units by looking at their stacked bar charts (Figure 4). For a better comparison between the probabilities assigned to each action, clicking on the charts to expand them to grouped bar charts.

By selecting a poor performing game from the game selector he can develop

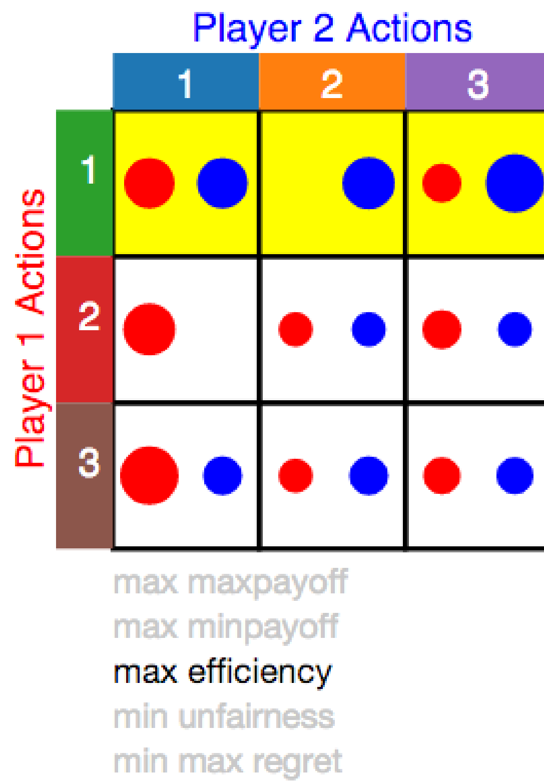

Figure 10: Hovering over a known hand-crafted feature, the row(s) corresponding to that feature light up. Non-hovered features are grayed out, to avoid distraction

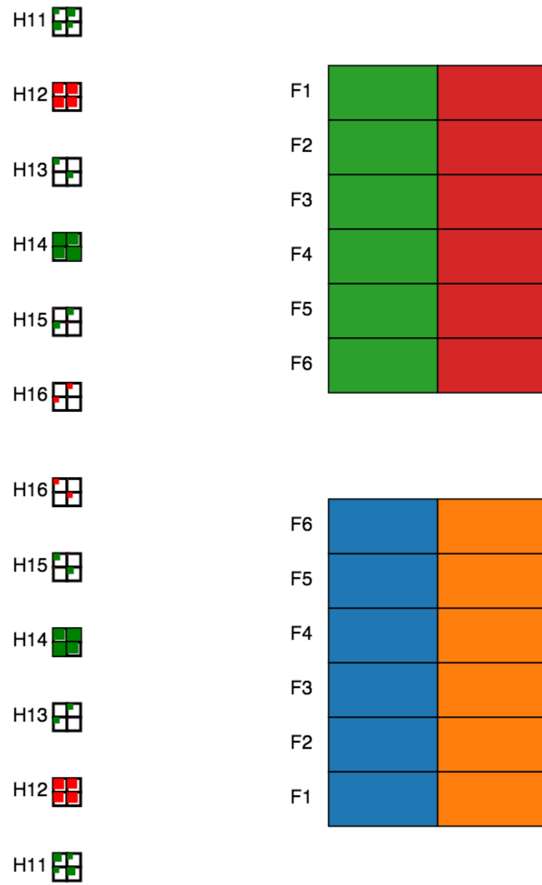

Figure 11: The visualization is flexible to changes in model parameters such as number of hidden layers or the size of each layer

hypotheses on what may be going wrong in his model. For example if a particular action is under-weighted, looking at which of the known features (Figure 10) perform suggest that action, may suggest that the feature is not being learnt by the model. This hypothesis could be strengthened by examining multiple games on which the model performs poorly to see whether or not the pattern is consistent.

6 Implementation

There are two components to our implementation: a Python-based API that handles the data and interactions with the model, and our front-end which is built using D^3 [1].

The Python API runs an implementation of the model and has access to all the data which allows it to deliver the model input and output data as well as any intermediate computation output that we need to visualise. This is then served to D^3 via Bottle [2], a simple Python web framework. It communicates the internals of the model to the visualization in JSON that is parsed on the client side. The visualization is therefore also protected against changes that the practitioner may make to the model: such as changing the number of hidden layers or action response layers.

The majority of our solution was built from scratch using standard D^3 components. The only exceptions to this D^3 Grid [10] which was used to draw the grid in the payoff matrices and hidden layers, and D^3 Tip [6] which was used to build the tool-tip for the hover action on the action response layers.

7 Discussion and Future Work

The major strength of *GameNetVis* is in providing an “at a glance” view of the outputs of nodes within the model on a particular game. Perfectly symmetric games with trivial outputs are easy to spot and oddities such as unusual feature output or hidden units that output zero values are equally clear with minimal effort on the part of the user.

Where it is less successful is in visualising the parameters of the network and showing weighted sums. The only parameters of the network that we explicitly show are those that make up the level distribution. While these are the most important parameters and easiest to interpret, there are a significant number of other parameters that we would have liked to show given the time to find an appropriate encoding.

We also would have liked to find an effective way of showing weighted sums. From a visualisation perspective, the model has an interesting property where there are multiple places in which you have multiple vectors that sum to one that are all added to make a new vector that sums to one.

Figure 12 shows an idea we had for representing the weighted sums that we ultimately decided was interesting but not practically useful for solving the tasks

presented. Further work could potentially find representations that exploit this property in a more useful fashion.

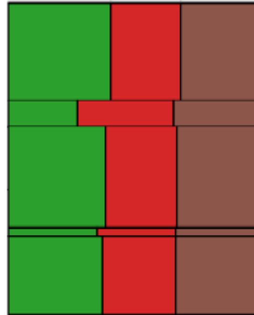

Figure 12: An idea for a glyph that shows the weighted contribution of outputs, where the weights sum to one. Here, each stack bar chart's height is proportional to the feature's associated weighting in the weighted sum.

Another idea that we experimented with and believe to be useful, but ultimately chose to leave out of this version of the visualisation was to build a method of exploring the model's output around a particular parameter setting. The model has approximately 150 parameters that define a function in a 150 dimensional space which clearly cannot be visualised. However, we had hoped to visualise the effect of changing each of the parameters independently while holding the others fixed. Figure 13 shows examples of these plots, which, given more time would have been a useful addition to the solution (albeit while solving a different problem to the ones described). Another similar idea was to show contour plots of multiple features against one another.

We also noticed that the blob payoff matrix encoding is invariant to scaling, so two scaled games look the same. This is unsatisfying given that we know humans have both a non-linear response to payoffs and a non-linear response to a particular visual encoding. A more principled visual mapping would attempt to find a visual encoding that is the closest match to the non-linearity in human response to payment such that one could get a feel for the size of a payment from the encoding of the payoff matrix directly.

Probably the weakest part of our project were the hidden layers. While better than nothing, they offer very little insight into what transformations to the data were leading to their outputs. A helpful simple improvement would to have a fish-eye (or similar) effect when hovering over a hidden unit to at least allow more detailed inspection of the size of the rectangles in the matrix. That said, it is not clear that the shaded matrix is the right visual encoding, or for that matter whether the outputs of the hidden layers provide insight into their function. An alternative approach would be to attempt to visualise parameters of the hidden layer rather than outputs, though it is unclear whether this would

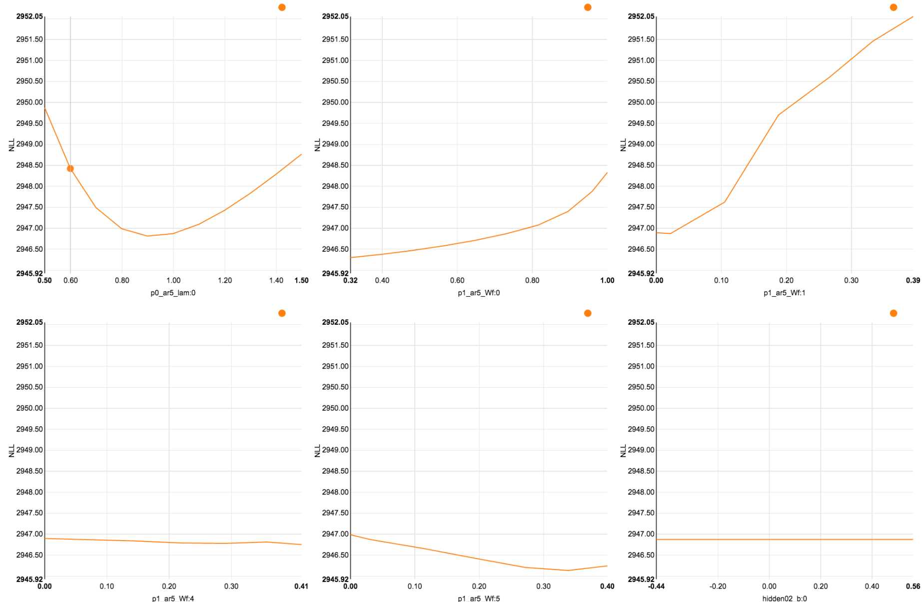

Figure 13: An earlier attempt at plotting marginals of parameters

be more helpful.

Finally, this implementation does not scale to large games. There exist games of 100 or more actions that would be useful to visualise. In order to do so it would be necessary to take advantage of the structure present in the description of the game rather than naively plotting 100×100 matrices. This is possible because these large games tend to have some structure encoded in them so that they may be described to humans in an understandable fashion. Finding a way to exploit this would be necessary for effective visualisation.

8 Conclusions

We have presented GameNetViz, a visualization for the GameNet model of normal form games. Our visualization lets the user observe the outputs at each computational unit, thereby allowing users to see how the output of a complex model was constructed. We hope that our tool will be useful for uncovering new insights as to what the model is doing and where it can be improved.

References

[1] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *Visualization and Computer Graphics, IEEE Transactions on*,

17(12):2301–2309, 2011.

[2] Marcel Hellkamp. Bottle: Python web framework. `http://bottlepy.org/docs/dev/index.html`, 2015.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[4] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 05 2015.

[5] T. Muhlbacher and H. Piringer. A partition-based framework for building and validating regression models. *Visualization and Computer Graphics, IEEE Transactions on*, 19(12):1962–1971, Dec 2013.

[6] Justin Palmer. D^3 tip. `https://github.com/Caged/d3-tip`, 2013.

[7] Harald Piringer, Wolfgang Berger, and Jürgen Krasser. Hypermoval: Interactive visual validation of regression models for real-time simulation. In *Computer Graphics Forum*, volume 29, pages 983–992. Wiley Online Library, 2010.

[8] Tom Schaul, Ioannis Antonoglou, and David Silver. Unit tests for stochastic optimization. *CoRR*, abs/1312.6055, 2013.

[9] Michael Sedlmair, Christoph Heinzl, Stefan Bruckner, Harald Piringer, and Torsten Moller. Visual parameter space analysis: A conceptual framework. *Visualization and Computer Graphics, IEEE Transactions on*, 20(12):2161–2170, 2014.

[10] Jeremy Stucki. D^3 grid. `https://github.com/interactivethings/d3-grid`, 2013.

[11] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

[12] T. Torsney-Weir, A. Saad, T. Moller, H.-C. Hege, B. Weber, J. Verbavatz, and S. Bergner. Tuner: Principled parameter finding for image segmentation algorithms using visual response surface exploration. *Visualization and Computer Graphics, IEEE Transactions on*, 17(12):1892–1901, Dec 2011.

[13] Fan-Yin Tzeng and Kwan-Liu Ma. Opening the black box-data driven visualization of neural networks. In *Visualization, 2005. VIS 05. IEEE*, pages 383–390. IEEE, 2005.

[14] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, volume 8689 of *Lecture Notes in Computer Science*, pages 818–833. Springer International Publishing, 2014.