Synthesizing baseball data with event prediction pretraining

Aren Dakessian
Center for Data Science
New York University
New York, NY
agd387@nyu.edu

Guido Petri
Center for Data Science
New York University
New York, NY
gp1655@nyu.edu

Rahul Zalkikar
Center for Data Science
New York University
New York, NY
rz1567@nyu.edu

Abstract

We train a neural network to predict baseball game events and use its features and predictions in the synthesis of baseball plays via a pre-existing network. Our game representations and game event predictions are inaccurate and worsen the pre-existing network's performance for small training procedures.

1 Introduction

Baseball has long been a very active area of sports analytics research, in part because of its natural propensity for analytics. In our project, we aimed to create a network that can synthesize baseball data using game events as a pretraining task. Having a network learn a representation of a game of baseball, as well as predict the next time steps of a game situation, indicates that the network will have learned how to play baseball. This would then allow us to conduct "what-if" analyses, such as what might happen if instead of a player being left-handed, a given situation happens with a right-handed player.

Our approach involved two steps: first, the modeling of game event prediction, and second, using our network's features and its game event predictions to enhance a pre-existing network that synthesizes baseball time series one step at a time.

2 Related Work

Ono et al. (2018) have already worked on this problem. Specifically, they created a graph variational recurrent neural network which can synthesize baseball plays given player covariates. For example, they can synthesize a play with a left-handed batter or a right-handed batter. However, the results indicate that their network sometimes enters a failure mode in which it does not produce realistic inputs. One such case would be that the network outputs a ball trajectory that does not follow the laws of physics, for example via a curve or momentum shift that is unexplained by a player's action.

Our work focuses on improving Ono et al. (2018)'s network via the addition of a pretrained network that provides additional context to their network. The idea behind this addition is that their network is not able to recognize when events happen, and so might predict a ball trajectory ignoring that a player might catch the ball, for instance. Using the context given by our network's outputs, we hope that their network synthesizes plays that look more realistic.

Additional work has been done in sequence generation in Graves (2014). The approach they use involves feeding a network its own inputs in order to generate the next step in a sequence. This was the approach taken by Ono et al. (2018) Our network would thus also undergo this procedure to find a sequence of game events for enhancing the dataset.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

3 Problem Definition and Algorithm

3.1 Task

The specific problem we attempted to solve was the generation of unlikely scenarios by Ono et al. (2018)'s network. We believe this was due to the network's representation lacking context of what actions players take, and thus being unable to change the player and ball trajectories due to those actions.

For example, some of the network's generated sequences are unrealistic due to the curvature of the ball trajectory, or the player trajectories changing directions too quickly for the momentum laws of physics.

Our output to this task was a vector that represents game actions, which we appended to Ono et al. (2018)'s network as features. In this scenario, we hope that the additional features in their network will help guide the network to more realistic scenarios that only have big momentum changes when events happen, for instance.

Our inputs were similar to those of Ono et al. (2018)'s inputs. We used tracking data from the *Baseball Metrics Engine*, developed by Prof. Claudio Silva at New York University, as well as tagged game events for this tracking data.

We approached this problem via pretraining, since we believe the network architecture that is used by Ono et al. (2018) is sufficient for the actual synthesis task. This has been demonstrated in Graves (2014), which used recurrent neural networks for both text and handwriting synthesis. In particular, their results differed greatly when using different kinds of input for their networks: when generating text, their performance increased greatly when using character-level one-hot-encoded vectors rather than word-level, and when generating handwriting sequences, the addition of accompanying text allowed the network to generate coherent handwriting rather than random strings of characters.

Similarly, we believe that the addition of game events to Ono et al. (2018)'s network should improve the generated baseball plays' plausibility. However, since these are synthetic plays, we are unable to dynamically provide labels to the plays via a human annotator. As a result, our idea was to train a network that is able to provide those game event labels dynamically.

Alternative approaches to solve this problem involve changing the generating network architecture or using a discriminator network in a fashion similar to how GANs are trained, for instance. We elected our pretraining approach since we judged it most likely to succeed with the limited neural network training resources we have at our disposal. The main disadvantage we see to our approach is that we are limited by some upper bound of the performance of Ono et al. (2018)'s network architecture, since we are only adding additional features to it.

3.2 Algorithm

We approached this as a classification problem. As such, we used both usual classification models such as logistic regressions and SVMs, as well as neural networks with appropriate classification architectures.

In particular, the architectures that we used were recurrent neural networks with LSTM or GRU type cells. We varied the depth of the network, as well as the number of hidden recurrent units in our experiments. We also used dropout in order for the network to have a self-regularizing effect, as well as a *feature norm*, in which we normalized each feature to have mean zero and standard deviation 1.

Additionally, we experimented with autoencoder-type recurrent neural networks, which are composed of an encoder and a decoder sub-network, each being a recurrent network. The architecture here was similar to the standard RNNs we used, in that we used dropout and feature norm.

Our main assumption in creating these network architectures is that the problem is difficult to solve without using previous inputs. Specifically, some of the events that are present in our target labels are impossible to classify with only a snapshot of information: *top speed*, for instance, cannot be predicted correctly without either speed or acceleration information, or accessible information from previous timestamps. As a result, we believed our recurrent networks would work far better than the standard classification models we used.

4 Experimental Evaluation

4.1 Data

For our experiments, we utilized tracking data for baseball plays that stems from the *Baseball Metrics Engine*, a project spearheaded by Prof. Claudio Silva. The data contains x, y, z coordinates for all 9 players on the field, as well as the batter, the ball and 3 possible runners, one on each base.

The x, y, z coordinates in the data refer to locations on the field. The x coordinate is along the line drawn between first and third base, and has its origin at the location of home plate. The y coordinate is along the line drawn between home plate and second base, and its origin is the very tip of home plate. Finally, the z coordinate is a height coordinate indicating how far above the field that an entity is. This means that the origin of our coordinate system is the back tip of home plate.

Additionally, the data contains labels for what events occurred at a given timestamp, as well as what player was involved in the event, or what player the event refers to. The full list of events is: *Begin of play*, *Pitcher's first movement*, *Ball was pitched*, *Ball was hit*, *Ball was deflected*, *Top speed*, *Ball was released*, *Ball was caught*, *Player reached 1st* (referring to first base), *Ball bounce*, *Tag was applied*, *Off the wall* (referring to the ball hitting a wall and bouncing off), *End of play*, and a *Not an event* marker. This totaled 14 distinct classes in our target distribution. These were not evenly distributed: the *Not an event* marker represented approximately 99.96% of our data rows, while, for example, the *Off the wall* event was only present twice in our dataset. We attempted to fix this severe class imbalance later, in our model training process.

The data is a sequence of timestamped coordinates for all relevant entities on the field: the ball and any players that are present. The polling frequency is different for different periods of the play, as well as for different entities. For instance, the ball locations are tracked only once the pitcher releases the ball, and then again every 3 milliseconds, whereas players are only tracked at 15 millisecond intervals. For players that were not on the field, such as runners who are not present at a given play, the data represents these as `None`. Additionally, for entities that are not being polled at a given timestamp, their coordinates are also represented as `None`.

Our preprocessing of the data was done in two steps. First, we had to clean the data; then, we had to format it appropriately for our models to ingest.

4.1.1 Data cleaning

Data cleaning was done by filling the missing values, as well as removing some features that were incompatible with Ono et al. (2018)'s network. The missing value imputation followed different rules for different columns, similarly to Ono et al. (2018). We also had to ensure our timestamps were consistent: for some of our datapoints, the initial timestamp was set to 0, but the following timestamps had a large, seemingly random number that increased at the correct polling rates.

The features we removed were all z coordinates for players, leaving only the ball with a z coordinate. This was done since players are expected to remain on the same z level, and not ascend or descend in height. Additionally, we removed the column showing which player was involved with an event, since it was missing a value if there was no event: this would induce leakage in our network, since we don't have that information available when generating a synthetic baseball play.

For the runner locations, missing values in the x coordinate were imputed with the value -100, and missing values in the y coordinate with 0. This imputation methodology was used by Ono et al. (2018), and so we had to follow it. However, the dataset we used did not have any runners in any of the plays, so we do not entirely agree with this imputation methodology - instead, we would have preferred to remove these features entirely.

For events, missing values were filled with the *Not an event* marker in order to make network prediction easier.

Finally, for timestamps that were missing values for specific entities, we interpolated the last known locations with the next known locations of that entity and used the interpolation to fill in the missing values. This method is strictly linear, and so induces some error into our data, but we estimate this error to be very small due to the fine-grained timestamps we have available to us. It is unlikely that a player accelerated and then decelerated in a 15 millisecond timeframe, for instance; rather, the player

most likely only had a small variation in speed, and so we think this interpolation method is valid. For any entities that were missing values in the beginning of the play, such as the catcher, we back-filled the first known location. The ball entity in particular was missing values before it was pitched, as well; since every play starts at a point where the ball is in the pitcher's hand, we used the pitcher's location coordinates to fill in missing values for the ball coordinates.

We extended the use of interpolation to interpolate entity locations at every millisecond in order to ensure polling consistency. Since events were also tagged at specific polling instances, we backfilled the events to ensure we did not assign the *Not an event* marker to actual events.

4.1.2 Input formatting

We tried two input formats for our data. First, what we termed *Last Few Values (LFV)*, and second, what we called *Full Play (FP)*. Additionally, we experimented with hand-crafted pairwise distance features between entities, as well as subsampling our data by only using every nth known location instead of using per-millisecond locations.

For the LFV format, we fed our network a sample that contained the current location of all entities, as well as some of their previous locations. This was intended to allow the network to learn speed representations, as well as movement direction. As a result, our network input was a tensor of shape

`(batch_size, num_samples, num_features),`

where `num_samples` is the number of samples of previous locations plus 1, for the current entity locations.

The FP format, on the other hand, used entire plays at once instead of just a handful of location polling samples from a play. Since different plays were of different length, we used zero-padding at the end of a play in order to ensure consistent tensor size. Our input tensors were thus of shape

`(batch_size, length_longest_play, num_plays, num_features).`

For this format, we also had to pad the targets, since there was a mismatch between the target length as well; these were padded with our *Not an event* marker.

The hand-crafted distance features were Euclidean distances between entities in a pairwise fashion. Additionally, we thought it would be useful to the network to introduce distances between each entity and each base. For the base locations, we used the documentation of *Baseball Metrics Engine*.

Our subsampling was done by selecting locations for our input with a fixed amount of time inbetween two sequential selections. This ensured polling consistency while reducing the amount of data we had for training and testing, and was useful in order to allow for faster model iteration and model training.

Additionally, we implemented a simple centering transform of our data. We subtracted the mean of each feature across all datapoints from each datapoint, using only the training set to determine the mean. The purpose of this transform was to allow the network to use only offsets from a center value rather than absolute values that are highly negative or highly positive. For example, it may be more interesting for the network to know that a left fielder was closer towards center field with a positive x value, rather than a "less negative" x.

4.2 Methodology

We split the data into training, validation and testing splits randomly based on play. We used a 60-20-20 split. These splits were saved to disk as binary files to make our data loading faster, since otherwise several individual play `.csv` files would be read each time the data was loaded into memory. Additionally, we saved our transformed versions of the splits to file, so that the feature engineering, subsampling and input formatting would not have to be repeated at every model training procedure.

For our experiments, we used individual class precisions as the metric to determine model performance. We judged precision to be a better metric than recall for this project, since we placed more importance on maximizing the number of events detected. For some of our experiments, we had to compare the maximum of class precisions and average of class precisions, as well, since it was difficult to compare models that performed well in predicting one event but poorly in another.

Since there have been no attempts at predicting these game events, we did not have a fixed goal for our precision metrics; rather, they guided us to find better models, not to decide when our model iteration was finished.

Model training was done using the ADAM optimizer with PyTorch's default momentum parameters of $(0.9, 0.999)$. We used PyTorch's built-in `CrossEntropyLoss` as our loss function. We attempted both using standard, uniform class weights, as well as weighing each class by a number proportional to the inverse of its incidence in the dataset.

For our models, we implemented deep-layer RNNs (`EventRNN`), a RNN with a linear featurization layer before the LSTM cells (`FeatureRNN`), and an auto-encoder RNN (`AutoEncoderRNN`).

The hyperparameters we explored were:

- Data input format
- Model architecture type
- Number of training epochs
- Batch size
- Learning rate
- Number of hidden RNN cell units
- Number of featurization units (for FeatureRNN)
- Number of layers
- Dropout probability
- RNN cell type (LSTM or GRU)
- Usage of feature norm
- Subsampling the data
- Centering the data
- Featurizing the data via the previously-mentioned distance measures

After training our models, we selected the best model and used it to dynamically provide labels to Ono et al. (2018)'s network by slightly modifying their network to have an additional 14 inputs in one of their linear layers and concatenating our network's outputs to their features. The synthesis network was then trained from scratch, with our pretrained event classification model in evaluation mode and with its weights frozen. We then generated 5 synthetic baseball plays to examine the Ono et al. (2018) model performance with and without our additional features. The same training procedure was used for both the original model and the modified model, and the same base plays were used to initiate the play synthesis.

4.3 Results

Our results suffered from the large class imbalance present in the data. A subsampling methodology to correct for that class imbalance is impossible, since we have sequential data that inherently has the class imbalance present in each play.

Across our experiments ($n = 43$), we had the class precision distributions shown in Figure 1. It is evident from the graph that the *Not an event* marker and the sheer count of these markers was a big driver in model performance. For the most part, our models had a precision for *Not an event* around 0.99. The next best class precisions were *Begin of play*, *End of play* and *Ball was pitched*.

Our best model was a 4 layer EventRNN with GRU memory cells, trained for 100 epochs with batch size 16 on data subsampled for 50 millisecond latencies, with distance featurization and feature norm. The following is a table of the five best models as measured by average class precisions:

In general, we found that:

- LFV format was far worse than FP
- GRU cells were better than LSTM

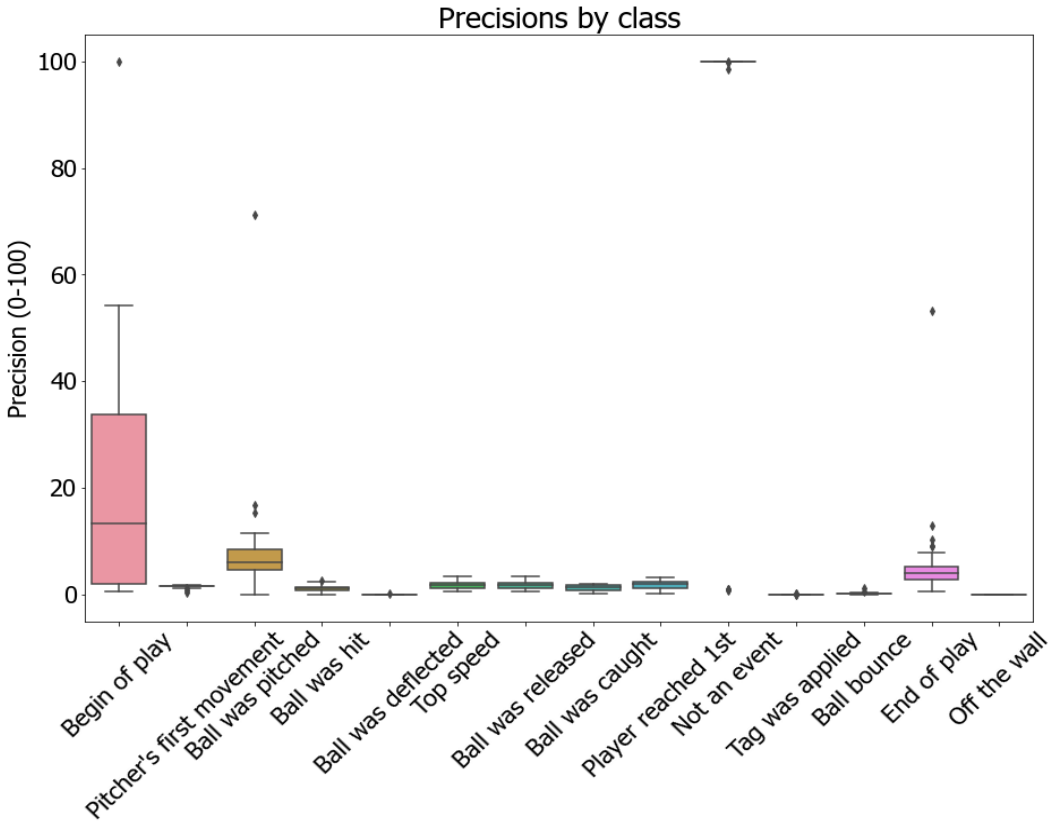

Figure 1: Precisions per class across all of our experiments.

Table 1: Best 5 networks by average class precision

Prince JobID	Model Description	Average class precision
13903970	FP, 4 layer GRU EventRNN, batch size 16, 100 epochs, 50ms subsampling, featurization, feature norm	0.1349
13890649	FP, 4 layer GRU EventRNN, batch size 16, 50ms subsampling, featurization, feature norm	0.1285
13899316	FP, 4 layer LSTM FeatureRNN, batch size 16, 50ms subsampling, feature norm	0.1282
13899272	FP, 4 layer LSTM FeatureRNN, batch size 16, 50ms subsampling	0.1274
13886485	FP, 4 layer LSTM EventRNN, batch size 16, 50 ms subsampling	0.1266

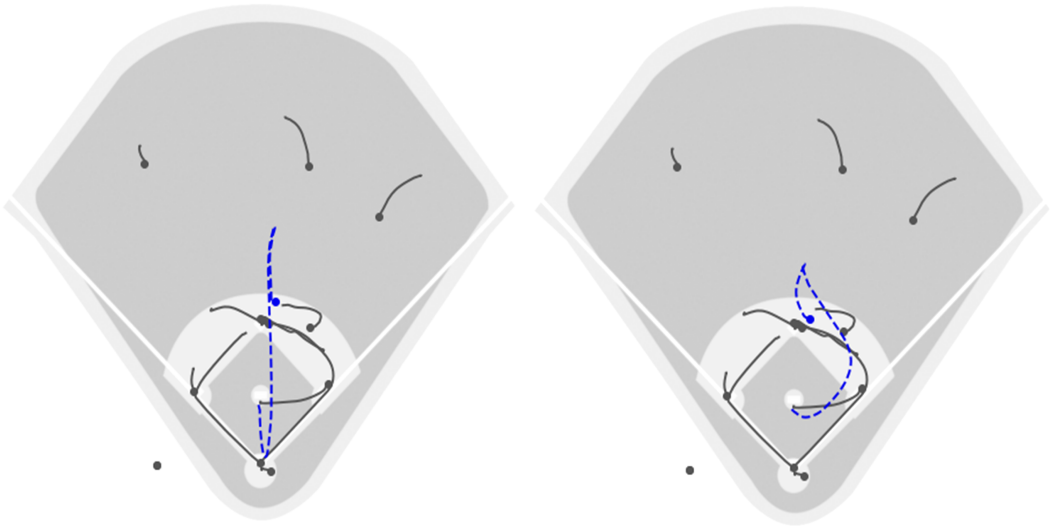

Figure 2: Play synthesis with the original network (left) and the modified network (right). The ball is in dashed blue, and players are in solid gray. The solid circles indicate last known positions.

- Feature norm greatly increased model performance
- Subsampling increased model performance
- EventRNN was our best model architecture
- Batch size and number of hidden units had very little effect on performance

Our full tabular results for the event classification models can be found in the attached spreadsheet file.

The synthesis network output was different in the original network and the modified network, though the modified network's output is less realistic than the original network's output. An example can be seen in Figure 2, and further examples can be found in the appendix.

4.4 Discussion

It's evident from the synthesis results that our model detracted from the original network's performance. The modified network had issues with the direction that entities moved in. In figure 2 above, this was displayed for the ball: the ball's trajectory curves in a way that is impossible for a player to pitch a ball.

The original network was only modified by addition of features, not by removal. As a result, our modified network should, in the limit, have performance at least equal to the original network, since the modified network could elect to ignore our custom features if they would worsen the network outputs. However, we observe that the results are worse. This could be explained by the additional features requiring the network to be trained for more epochs, or with more data, to achieve the same performance as the original network.

Additionally, we believe that the classification results can be improved substantially. The main classes that had good precision for our classification network all have easily explained or easily found ways of predicting them: *Begin of play* can be found by noticing all the hidden units' values are at their initial value of 0; *Ball was pitched* can be found by measuring distance between the ball and the pitcher, and claiming the ball was pitched when this distance is nonzero; *End of play* can be found relatively easily by measuring distance between the ball and first base, or the batter and first base, and claiming this class when either distance is zero. It's unclear why the classification network was unable to learn these very simple rules for the above classes, or any other classes.

Performance for our network greatly increased when we subsampled the data. This supports the theory that the network's low performance is largely the effect of the severe class imbalance present

in the data. However, we do not have good suggestions for how to improve the classification accuracy. Should classification performance improve, we hope that the additional features made available to the synthesis network will still improve the synthesis network.

5 Conclusions

Our classification networks did not achieve very good per-class precisions on the pretraining task. This poor performance carried over to the modified synthesis network, where our features detracted from the original synthesis network's performance.

We believe our modified network detracted from the original network solely due to the poor classification performance. In future work, should this event classification performance be better, then we believe the additional features would help the synthesis network generate more realistic plays.

6 Lessons learned

We learned several lessons during this project: how to work with RNNs and autoencoders, working with large datasets, the importance of data input format, the potential gains of data subsampling, and how to expand pre-trained architectures and use pretraining in new neural networks.

None of us in the project group had a good grasp of how RNNs and autoencoders worked. Implementing these architectures ourselves, even if with the help of PyTorch, helped us understand how each network type works and what its advantages and disadvantages are.

Given the size of our dataset, we had to be careful with how we manipulated it. We learned when it's useful to keep a dataset in memory, as well as when to write a dataset to file for later loading.

We learned how important the data input format is for a neural network. Our LFV format had extremely poor performance, whereas the FP format had a performance orders of magnitude greater. We recognize that neural networks are extremely powerful, but if we can impose structure upon the input data, we can help networks learn faster.

We subsampled our data to increase the potential number of epochs trained. This led to a great advance in our precision metrics, and, in fact, the subsampled data led to our best network. We believe subsampling data like this and then fine-tuning on a larger subset of data may be a good technique to carry forward to other projects that involve neural networks.

Finally, we learned how to use pretraining and how to extend existing neural network architectures, as well as the potential advantages of such approaches.

Acknowledgments and Disclosure of Funding

We give our heartfelt thanks to Jorge Ono and Prof. Claudio Silva, who kindly provided us with the resources and feedback we needed to complete this project.

References

[1] Ono, Jorge Piazentin, Dey, Neel, Lourenco, Raoni de Paula (2018). MLB Game Synthesis. Project Report for NYU's CS-GY 9223 class.

[2] Graves, Alex (2014). Generating Sequences with Recurrent Neural Networks. Arxiv.

Student contributions

- Aren: model training, hand-crafted features for classification, cluster environment setup, reproducing original network for synthesis
- Guido: data loader, data formatting, project management, model integration, report
- Rahul: model building, model training, model evaluation, data featurization

Appendix

Github Repository

Our code can be found under:

`https://github.com/cs3026/nyu-cds-capstone-baseball-2020`

Synthesis results

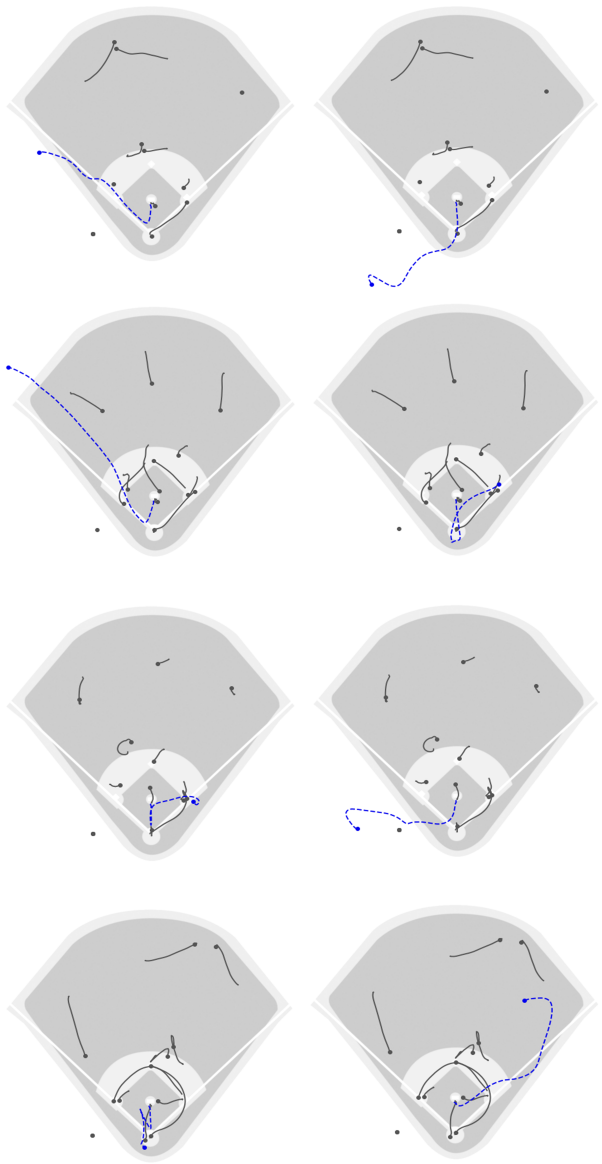

Figure 3: More examples of play synthesis with the original network (left) and the modified network (right).