HANDS-ON TUTORIALS

Teaching a Neural Network to Play Cards

How I trained a neural network to play a trick-taking card game without requiring human input

Michael Wurm · Follow
Published in Towards Data Science
9 min read · Apr 25, 2021

I picked a card game that I liked playing while growing up and my goal was to develop a system that can teach itself without human interaction and to arrive at a model that is good enough to beat my father.

Image by Author, derived from English Pattern Playing Cards Deck (public domain) by Dmitry Fomin and Table Texture (CC BY 3.0 License) by ScooterboyEx221

The game is known in Austria and Bavaria as “Grünober Suchen” or “Grasobern”. I’ll just call it “Queen of Spades” here and spare you the introduction to German cards. I chose this game because the difficulty level is just about right so that it is easy to learn but it still has some twists that make it interesting.

The result of the project is an Android app where you can play against AI-based opponents with adjustable difficulty level.

How to Play

Queen of Spades is a trick-taking card game that is related to Hearts. The game is for 4 players and is played with a deck of 32 cards. There are four suits: Hearts, Diamonds, Spades, and Clubs. Each suit has the following ranks in descending order: Ace, King, Queen, Jack, 10, 9, 8, and 7.

Each player receives 8 cards. The first player leads the trick by placing a card face up on the table. The other players each follow with a single card, in clockwise direction. They must follow the suit of the first card, if possible. If not possible, they can play any card. The trick is won by the player who played the highest card of the suit matching the first card. The winner takes the cards, places them face down on a pile, and leads to the next trick.

The goal is to avoid winning the trick that contains the queen of spades, and to avoid winning the last trick. Therefore, the normal strategy is to discard high-value cards early. But as an added twist the game can also be won by winning all tricks, which requires a completely different strategy. The decision to go for all tricks is usually made dynamically in the middle of a game. It doesn’t need to be announced and often comes as a surprise to other players.

A more detailed explanation, including some strategies, can be found here.

Scoring

Points are distributed so that the sum of all points is zero:

- A player who wins all tricks gets +3 points, all others get -1 point.
- A player who wins the trick with the queen of spades *and* wins the last trick (but not all tricks) gets -3 points, all others get +1 point.

- Otherwise, the player who wins the trick with the queen of spades gets -1 point, the player who wins the last trick gets -1 point, and the other two players each get +1 point.

Training the Model

My goal was to set up a system that can teach itself without human interaction. I started by implementing the framework for the game in Python, where four simulated players can play against each other. Initially, lacking a trained model, the players select cards randomly, out of the choices that are permitted by the rules.

Each game has 8 rounds, but in the last round, with only one card left, there is really no decision to be made. For each game I was able to collect information about 28 situations and moves (7 rounds * 4 players).

Defining the Input and Output Vectors

The input vector needs to describe the current situation of the game with an adequate level of detail. To be fair, only information that is available to the current player can be used. However the player is able to count cards.

I chose this representation with a total of 115 elements:

- Number of the round being played (8)
- Cards in hand by active player (32)
- Cards on table (32)
- Combination of all cards in other players' hands (32)
- First suit played (4)
- Order of current player; equivalent to number of cards on table (4)
- Flag: Current player has not won any tricks so far (1)
- Flag: Current player has won all tricks so far (1)
- Flag: Current player has won the trick with the queen of spades (1)

The numbers in parentheses show the number of elements used to encode each component. Everything is one-hot encoded with either `-1` or `1`. For example the round number would be encoded as `[-1, 1, -1, -1, -1, -1, -1, -1]` in the second round. Zeroes are used if something is unknown or doesn't apply (for example the first suit played before a card has been played).

The output vector is simply a one-hot encoding of the card to play with 32 elements. The idea is that after inference the legally playable card with the highest value in the output vector is the best card to play.

Training Data

As I pointed out above, each simulated game produces 28 rows of training data, 7 rows for each player. After the game, each player gets a score, which can be -3, -1, +1, or +3 points. Each row consists of the input vector that describes the situation of the game, and an output vector. The output vector is filled with zeroes, except *the element representing the card that was played is set to the score the player received at the end of the game*.

For example, the data generated by a player that received -3 points in the end might look something like this:

```
Round  Input Vector                  Output Vector
    1  [1,-1,-1,-1,-1,-1,-1,-1...] [0,0,0,0,0,-3,0,0,0,0,0...]
    2  [-1,1,-1,-1,-1,-1,-1,-1...] [0,0,0,0,0,0,0,0,0,-3,0...]
    3  [-1,-1,1,-1,-1,-1,-1,-1...] [0,-3,0,0,0,0,0,0,0,0,0...]
...
```

In this example the player had played the card represented by element 6 in the first round, 10 in the second round, and 2 in the third round.

Using this approach the model should learn that it is a bad idea to play those cards in the given situations because they led to a negative score in the end.

However, a more nuanced approach is required because when losing a game, not all rounds are always contributing equally to the loss. For example: A player wins the trick with the queen of spades in round 3, but then avoids winning the last trick. In this case

only rounds 1 through 3 contributed to the negative score while the other rounds were actually good because they prevented an even worse outcome.

Loss Function

There is a problem when using the standard mean-squared-error as loss function during training.

Consider this output vector: `[0,-3,0,0,0,0,0,0,0,0,0...]`.

The model will learn that playing card #2 is bad, but it will also learn that playing any other card results in a neutral outcome. In reality we don't have any information about what would have happened if the player had played any other card in the same situation.

Therefore, zeroes in the output vector should not be used for training because they indicate that there is no data, and not that the target value is zero. For that reason I used the following custom loss function that makes sure that back-propagation only happens for the non-zero element:

```
def squared_error_masked(y_true, y_pred):
    """ Squared error of elements where y_true is not 0 """
    err = y_pred - (K.cast(y_true, y_pred.dtype) * scale_factor)
    return K.sum(K.square(err) * K.cast(K.not_equal(y_true, 0),
        y_pred.dtype), axis=-1)
```

Network Structure

The exact structure of the neural network turned out to be not very important. I have had good results with this and similar structures:

```
layers = [  # input_shape=(115,)
        [(384,'elu')],
        [(384,'elu')],
        [(256,'elu')],
        [(128,'elu')],
        [(32,'tanh')]
    ]
```

Evaluation of a Model

In order to evaluate a model I let it compete in simulated games against random players, or against players controlled by a different model (normally two players each). After several thousand games, I compare the average scores of both sets of players to determine which ones play better and by what margin.

Iterative Training

It turns out that using training data only from random games doesn't lead to a model that plays very well. For example, it is very unlikely that a player wins all tricks when everybody just plays cards at random. The model therefore doesn't learn how to win in that way.

It is better to use an iterative approach, where the initial training round uses just random data, but the next set of training data is produced by having the resulting model play against itself or against random players.

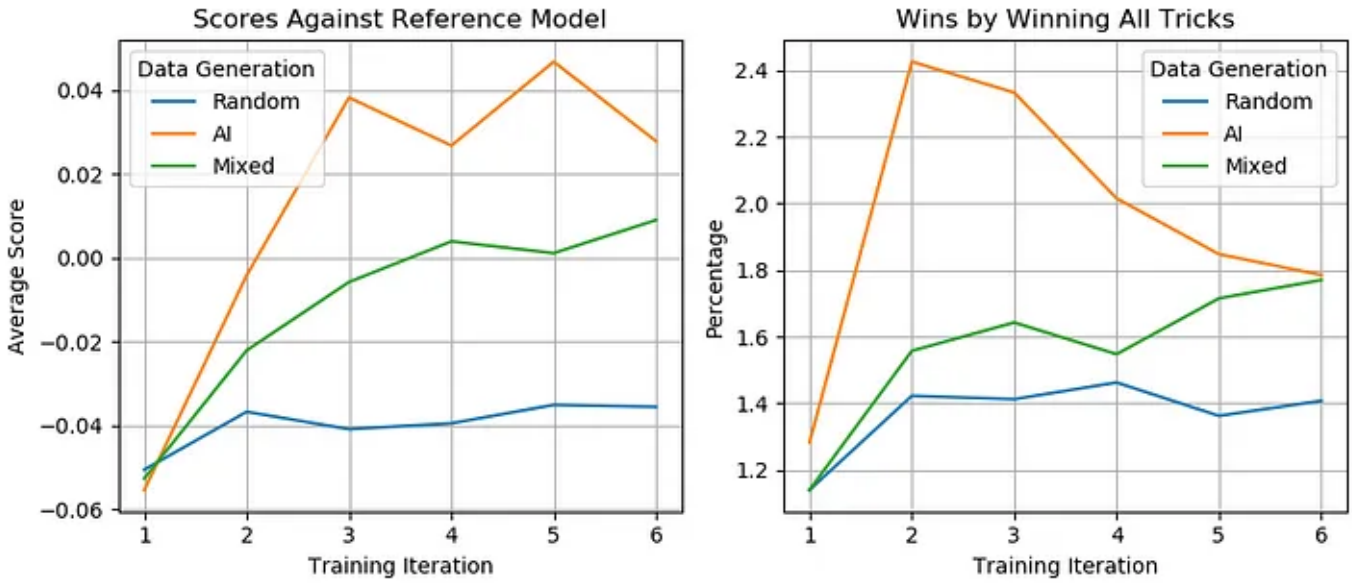
Algorithm

1. Generate training data from random games.
2. Train a new model using that data.
3. Evaluate the new model against the current best model. If the new model is better, then save it to replace the best model.
4. Quit after a set number of iterations or if there hasn't been any improvement for a while.
5. Generate new training data from games where the current model plays against itself.
6. Continue to train the current model with the newly generated data for a few epochs.
7. Go to step 3.

The data set each time was based on 1 million games (28 million records). If memory use is not a concern then it could also make sense to add new training data in step 4 instead of replacing the existing data.

Results

The following chart shows how the performance of the model-in-training progresses with each iteration when competing against a previously trained reference model (two players each). For reference I used the model that is currently deployed in the app.



Progression of Performance During Iterative Training (Image by Author)

Different methods to generate additional training data in step 5 of the algorithm are compared:

- Random: Data from games where players make random moves.
- AI: Data from games where the model plays against itself.
- Mixed: Data from games where the model plays against random players (two AI players and two random players).

It is apparent that the approach where data is generated from games that are played intelligently leads to better outcomes.

The percentage of games that are won by winning all tricks is significantly higher as well. It shows an interesting progression where it peaks after round 2 and then declines.

A possible explanation: After round 1 the model knows the normal strategy, which is to discard high-value cards early. Playing that way actually makes it easier to win all tricks

because later in the game, high-value cards that could be used to prevent a player from winning all tricks are no longer in play. In subsequent iterations the model adapts, reducing the likelihood that an attempt to win all tricks is actually successful.

TensorFlow Performance

While simulating games to evaluate models or to generate training data it is necessary to invoke the TensorFlow model on one single game situation at a time. TensorFlow is optimized to process large data sets simultaneously and has a substantial overhead when used on individual data records. It turns out that by converting the model to TensorFlow Lite, a tremendous increase in speed can be achieved.

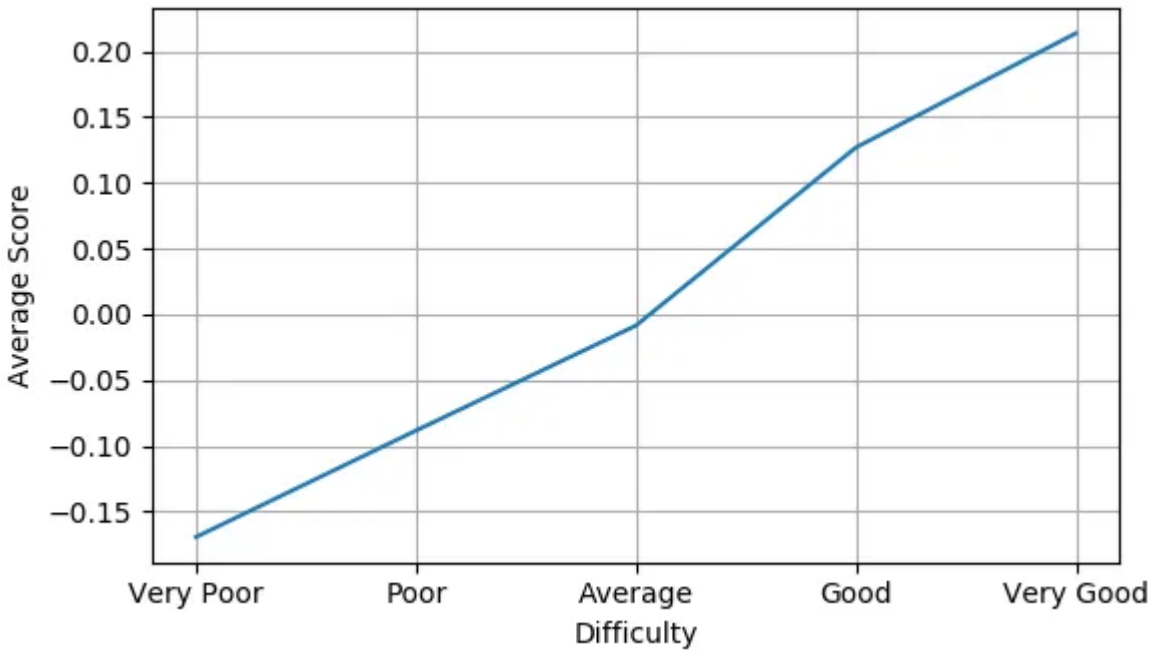
See also: "Using TensorFlow Lite to Speed up Predictions".

Adjusting the Difficulty Level

In the app I implemented five settings for difficulty:

1. Very Good: The model runs unmodified. The AI players are allowed to count cards, which is implemented by providing the "Combination of all cards in other players' hands" as part of the input vector.
2. Good: Card counting is limited. The cards of the suit of spades are still fully counted, but all other cards are counted simulating imperfect memory. I implemented this by scaling in half the appropriate elements of "Combination of all cards in other players' hands" so that they have less impact on the result.
3. Average: No card counting. All elements of "Combination of all cards in other players' hands" are set to zero.
4. Poor: No card counting and errors are introduced. Noise is added to the output vector before the best legal move is determined.
5. Very Poor: No card counting and more errors are introduced. More noise is added.

The following chart shows how difficulty levels compare. Two players with varying difficulty compete against two players on the average setting. The average score of the former players is shown.



Comparison of Difficulty Levels (Image by Author)

Conclusion

The test against humans is ongoing. If you want to see how you compare, you can get the game here: Queen of Spades on Google Play (free and no ads). As far as my father is concerned — he is in trouble when playing opponents on the Very Good level. But he can beat them on the Good level and eventually wound up with a positive score after hundreds of games.

AI | Neural Networks | TensorFlow | Editors Pick | Hands On Tutorials

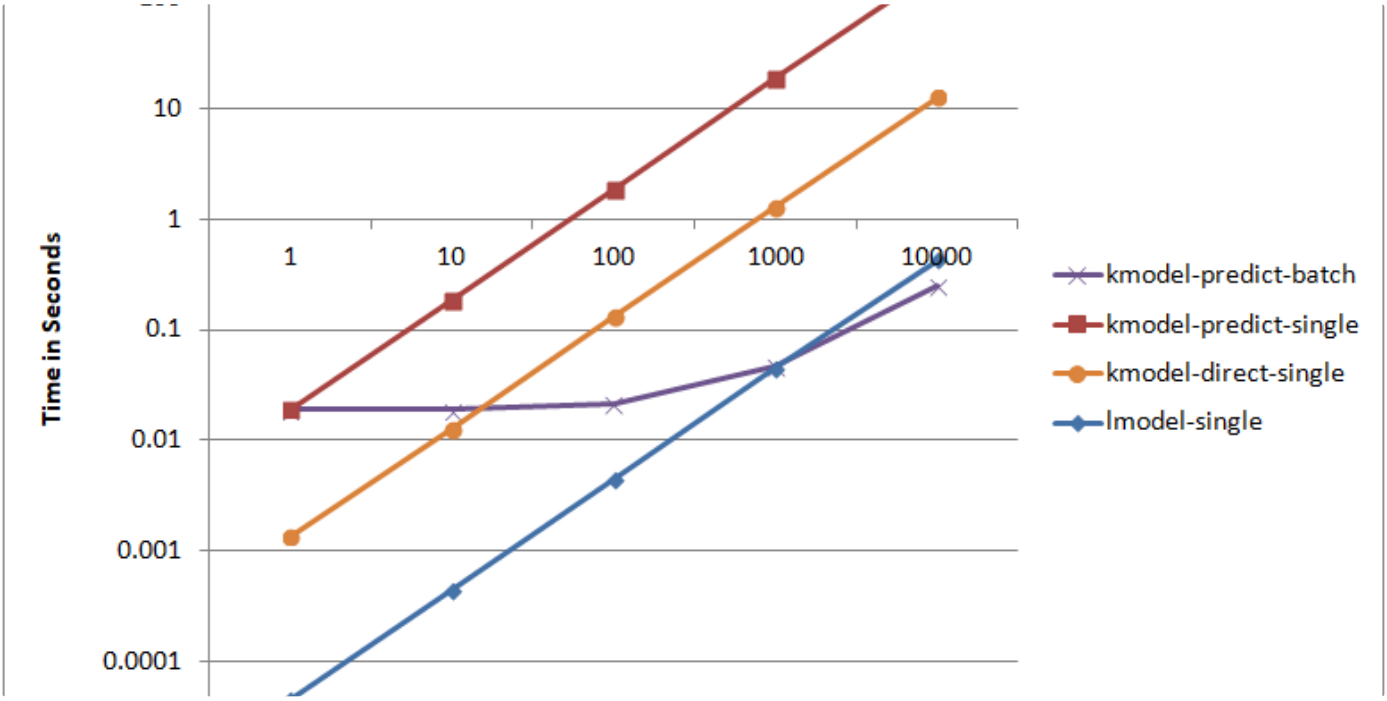
Follow

Written by Michael Wurm

31 Followers · Writer for Towards Data Science

Metabolizing caffeine into high-quality code, circuits, and mechanical designs

More from Michael Wurm and Towards Data Science



Michael Wurm

Using TensorFlow Lite to Speed up Predictions

Speed up predictions on individual records or small batches by converting a TensorFlow/Keras model to TensorFlow Lite

2 min read · Mar 9, 2020

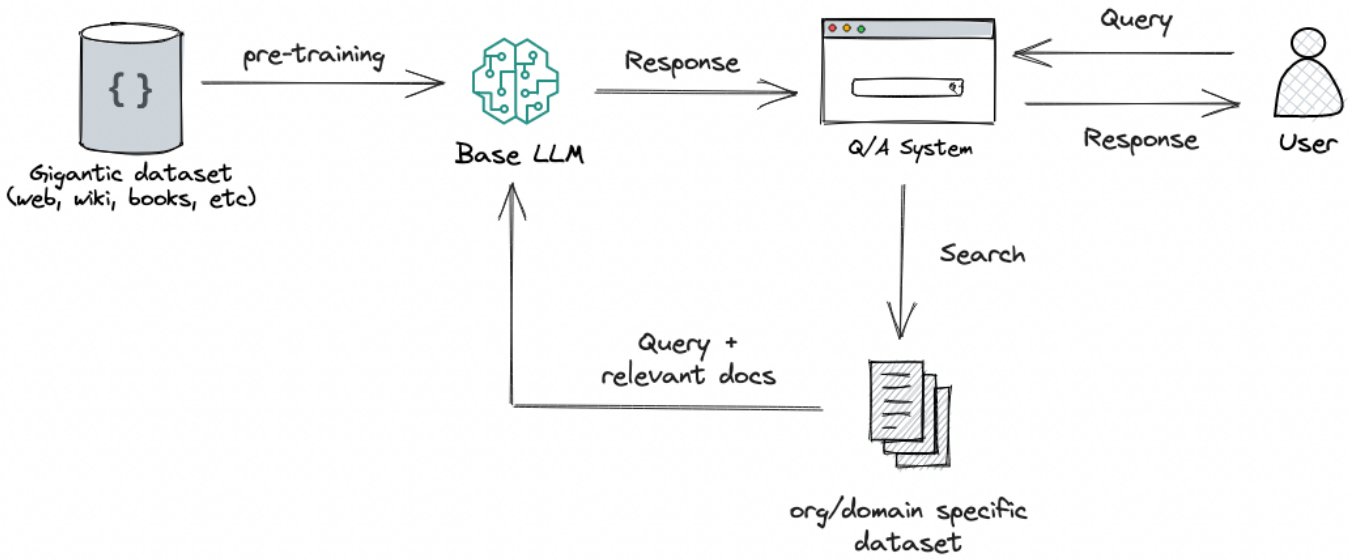
9 1

Giuseppe Scalamogna in Towards Data Science

New ChatGPT Prompt Engineering Technique: Program Simulation

A potentially novel technique for turning a ChatGPT prompt into a mini-app.

9 min read · Sep 3



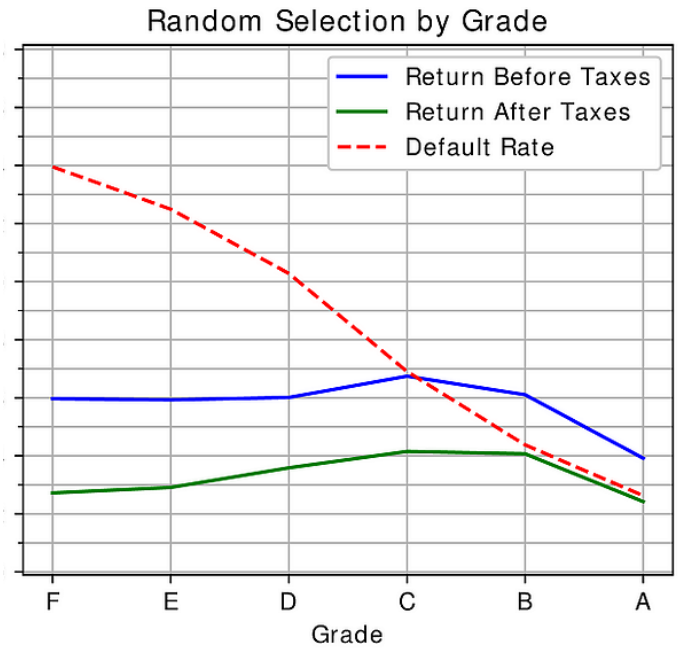
Heiko Hotz in Towards Data Science

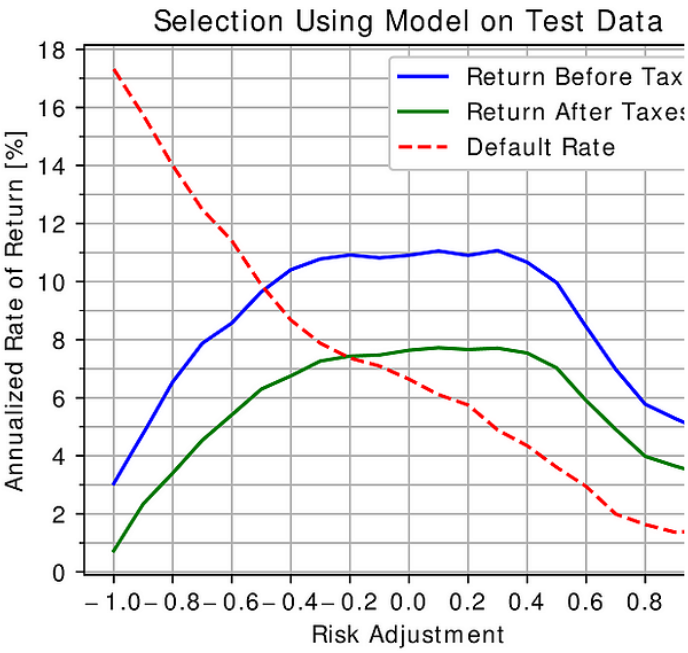
RAG vs Finetuning—Which Is the Best Tool to Boost Your LLM Application?

The definitive guide for choosing the right method for your use case

· 19 min read · Aug 24

2.2K 16





Michael Wurm in Towards Data Science

Intelligent Loan Selection for Peer-to-Peer Lending

How to use a neural network to pick loans on Lending Club while adjusting the risk in loan selection, and how to set up automatic...

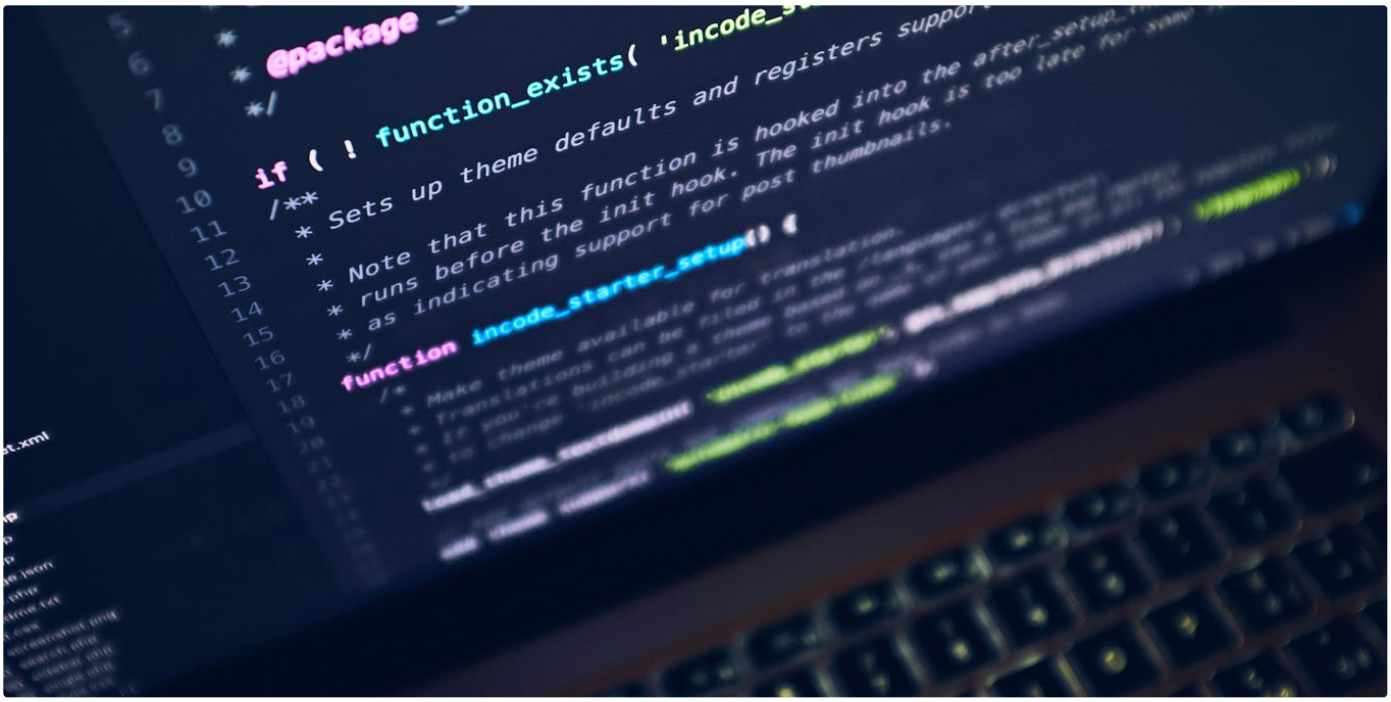
22 min read · Aug 19, 2019

56

See all from Michael Wurm

See all from Towards Data Science

Recommended from Medium



Zahrizhal Ali

Crafting Your Custom Text-to-Speech Model.

Welcome to a wild journey where code and comedy collide! In this blog, we're going to embark on a whimsical quest to build our very own...

11 min read · Jun 2

194 2

Felipe Jeon

Gentle introduction to Hybrid A star

Hybrid A* is considered one of the most popular path planning algorithms for car-like robots. It was developed by Dmitri Dolgov, who...

10 min read · Jul 23

16 1

Lists

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 127 saves

Generative AI Recommended Reading

52 stories · 263 saves

What is ChatGPT?

9 stories · 180 saves

Natural Language Processing

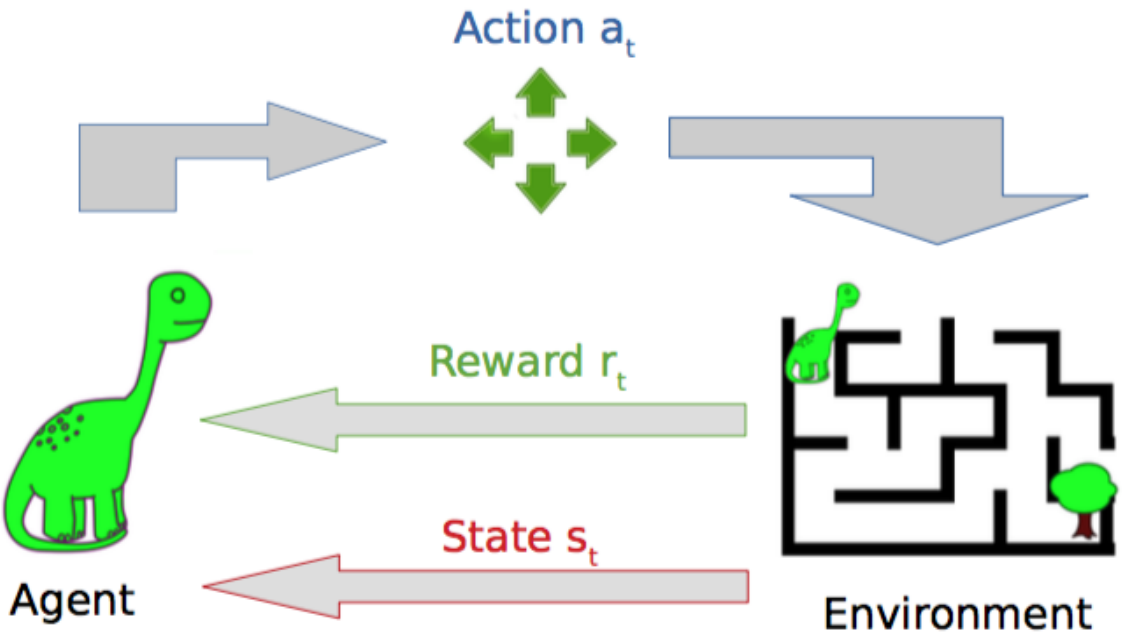
657 stories · 261 saves

Eivind Kjosbakken in Dev Genius

Creating an AI chess engine: Encoding using the AlphaZero method

Story overview

· 8 min read · Jun 21



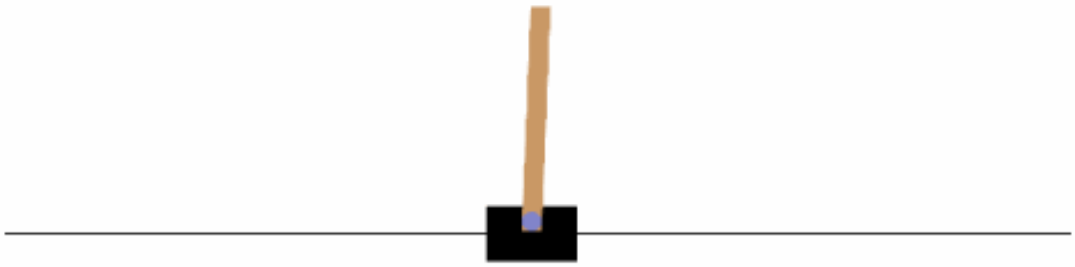
BAHADIR ARABACI

The reward hypothesis

In reinforcement learning, the learning process typically follows a loop that generates a sequence of state-action-reward-next state...

2 min read · May 31

8



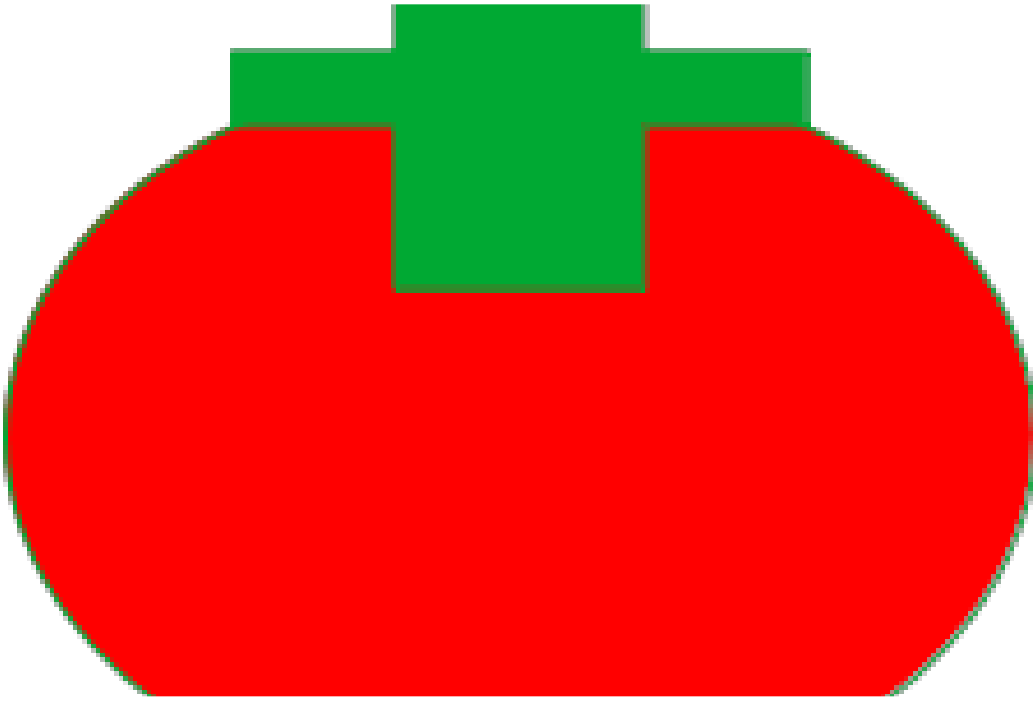
Ludovico Buizza

Solving the CartPole Environment

A (brief) introduction to reinforcement learning, in plain English.

10 min read · Apr 10

16



Douglas Cardoso in Better Programming

Run Asynchronous Calls Against WebDrivers

Introducing Caqui: a powerful tool for web scraping that is constantly evolving

4 min read · Jun 13

160 4