Baseball Strike Predictor

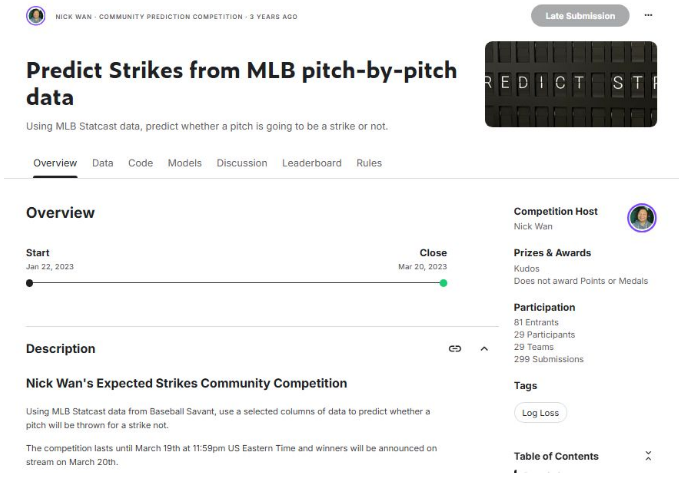
Ryan Don, David Garcia
Group ID: 29

The Problem

We were interested in doing a project involving baseball, specifically predicting something with pitch data.

Major League Baseball (MLB) data is very rich, thus we thought it would be a good idea.

Community challenge on **Kaggle**, Predict Strikes from MLB pitch-by-pitch data



NICK WAN · COMMUNITY PREDICTION COMPETITION · 3 YEARS AGO

Late Submission

Predict Strikes from MLB pitch-by-pitch data

Using MLB Statcast data, predict whether a pitch is going to be a strike or not.

Overview

Start	Close
Jan 22, 2023	Mar 20, 2023

Description

Nick Wan's Expected Strikes Community Competition

Using MLB Statcast data from Baseball Savant, use a selected columns of data to predict whether a pitch will be thrown for a strike not.

The competition lasts until March 19th at 11:59pm US Eastern Time and winners will be announced on stream on March 20th.

Competition Host
Nick Wan

Prizes & Awards
Kudos
Does not award Points or Medals

Participation
81 Entrants
29 Participants
29 Teams
299 Submissions

Tags
Log Loss

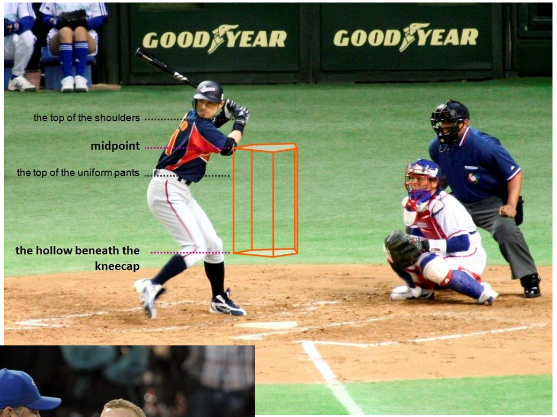
Table of Contents

The Problem

The goal is to predict if pitches are called **strikes or balls** by the umpire using **MLB Statcast** data.

A pitch be **objectively called** a strike if it touches the invisible strike zone, however, the ultimate call is **up to the umpire**.

Since there is a human element, this has some variance from the truth…



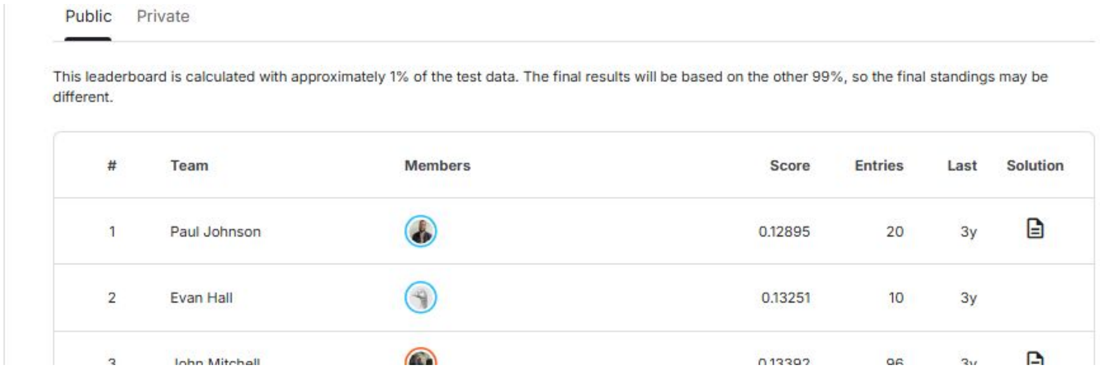
*here is a link to the official Major League Baseball Rulebook
https://www.mlb.com/glossary/rules/strike-zone

The Problem

In this challenge evaluated with log loss, not just simple accuracy.

The model needs to not only be correct with it’s predictions, but it needs to be confident as well. More importantly, if it’s very confident, it better be right!

This provides value when pitchers can identify certain areas outside the zone where pitches are frequently called strikes, depending on the game state.



Public Private

This leaderboard is calculated with approximately 1% of the test data. The final results will be based on the other 99%, so the final standings may be different.

#	Team	Members	Score	Entries	Last	Solution
1	Paul Johnson		0.12895	20	3y	📄
2	Evan Hall		0.13251	10	3y	
3	John Mitchell		0.13392	96	3y	📄

Our Solution

- A Multilayer Perceptron neural network was built with pytorch and lightning to classify pitches based on trajectory & game context

- A preprocessing pipeline was built to handle feature selection, encoding, and data scaling for both training and testing/submission datasets

- The model outputs raw logits which are converted to probabilities via a sigmoid function. These probabilities are clipped between 0.01 and 0.99 to avoid log loss penalties for confidently incorrect predictions

Data Used

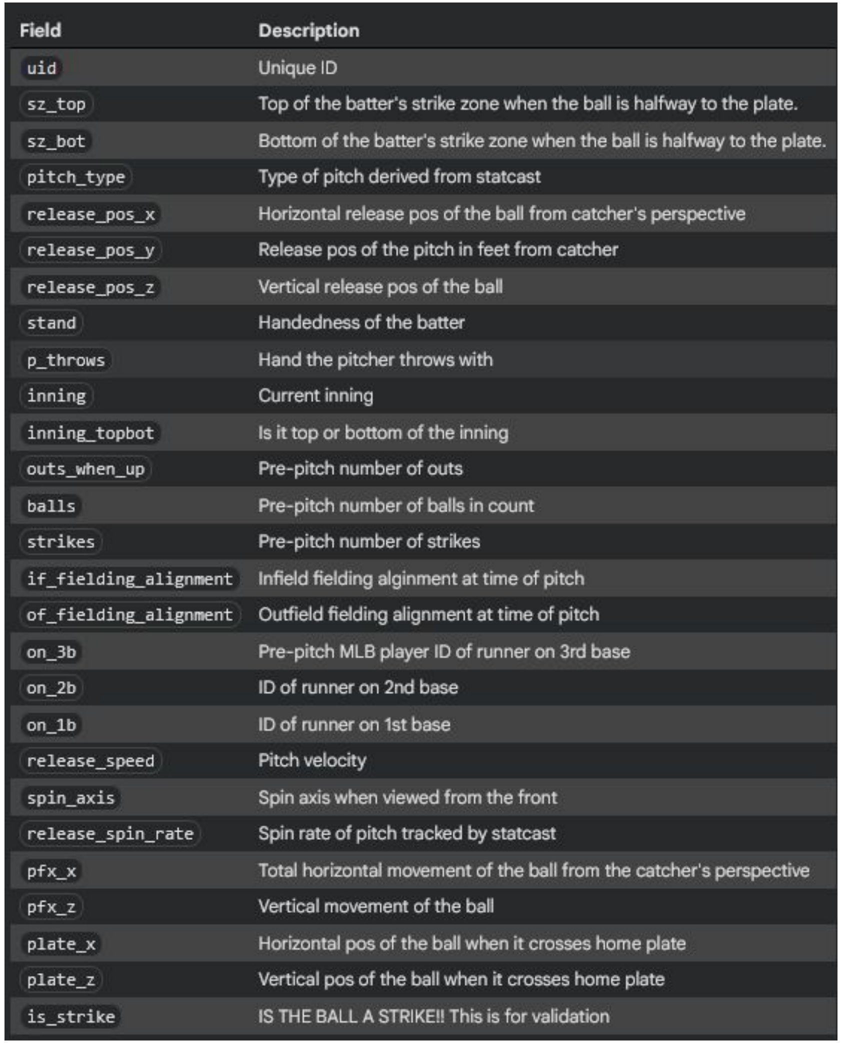
This project uses the dataset from the kaggle Predict Strikes from MLB pitch-by-pitch data community competition, hosted by **Nick Wan**

The data comes from the 2022 regular season and has many columns both useful and not.

There are about 300k rows to train on and 25 input columns. There is only one binary output is_strike.

The data has been split into two csv files train.csv and test.csv where we upload a file with all the probabilities output from test.csv to see how we placed.

Data Used



Field	Description
`uid`	Unique ID
`sz_top`	Top of the batter's strike zone when the ball is halfway to the plate.
`sz_bot`	Bottom of the batter's strike zone when the ball is halfway to the plate.
`pitch_type`	Type of pitch derived from statcast
`release_pos_x`	Horizontal release pos of the ball from catcher's perspective
`release_pos_y`	Release pos of the pitch in feet from catcher
`release_pos_z`	Vertical release pos of the ball
`stand`	Handedness of the batter
`p_throws`	Hand the pitcher throws with
`inning`	Current inning
`inning_topbot`	Is it top or bottom of the inning
`outs_when_up`	Pre-pitch number of outs
`balls`	Pre-pitch number of balls in count
`strikes`	Pre-pitch number of strikes
`if_fielding_alignment`	Infield fielding alignment at time of pitch
`of_fielding_alignment`	Outfield fielding alignment at time of pitch
`on_3b`	Pre-pitch MLB player ID of runner on 3rd base
`on_2b`	ID of runner on 2nd base
`on_1b`	ID of runner on 1st base
`release_speed`	Pitch velocity
`spin_axis`	Spin axis when viewed from the front
`release_spin_rate`	Spin rate of pitch tracked by statcast
`pfx_x`	Total horizontal movement of the ball from the catcher's perspective
`pfx_z`	Vertical movement of the ball
`plate_x`	Horizontal pos of the ball when it crosses home plate
`plate_z`	Vertical pos of the ball when it crosses home plate
`is_strike`	IS THE BALL A STRIKE!! This is for validation

Data Used

Inputs:

- Plate x, Plate Z (Position of the ball when it crosses home plate/the strike zone)
- SZ Top, SZ Bottom (Top and bottom of the batter’s strike zone)
- Balls, Strikes (How many of each there are when the pitch was thrown)
- Release Speed (Speed of the thrown pitch)
- PFX X, PFX Z (Horizontal and vertical movement after the pitch is thrown)
- Dominant hand of pitcher and batter respectively (Initially L,R had to be encoded)

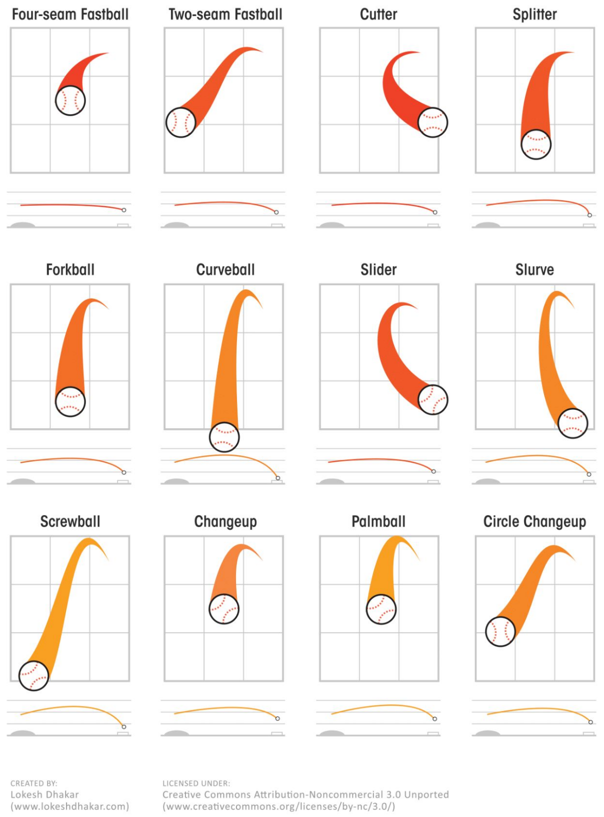
Output:

- Is strike (How confident our system is that the umpire will call a strike)

Data Used

Some examples of data we did not use and why:

- Pitch type: data already available in pfx and release speed
- Spin: already reflected in pfx
- On 1/2/3b: other umpires are responsible for watching these players

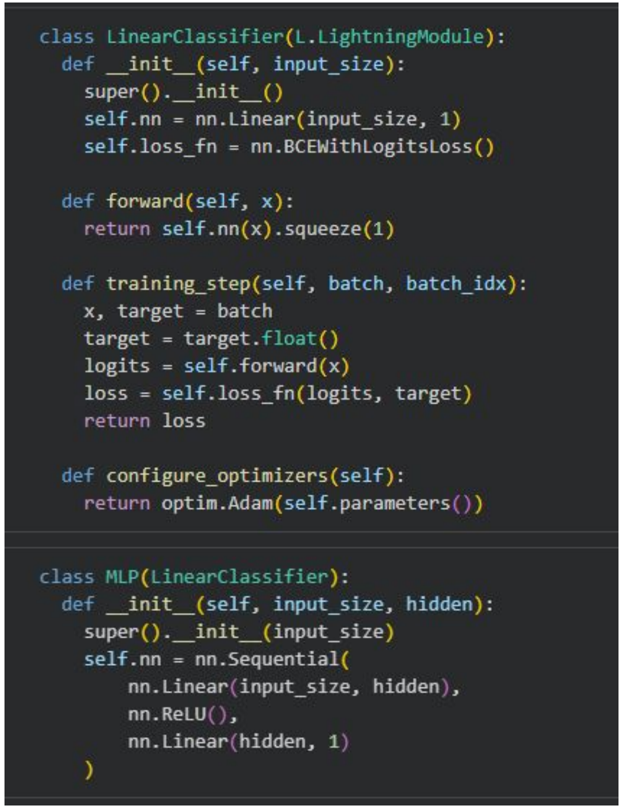


Initial Network Design

- Started with Lab 5 code, initially testing with a Linear Classifier, then overrode with an MLP class

Architecture

- **Input Layer**: 11 features
- **Hidden Layer**: Linear layer with 64 neurons followed by a ReLU activation
- **Output Layer**: Linear layer condensing to a single raw logit



```
class LinearClassifier(L.LightningModule):
  def __init__(self, input_size):
    super().__init__()
    self.nn = nn.Linear(input_size, 1)
    self.loss_fn = nn.BCEWithLogitsLoss()

  def forward(self, x):
    return self.nn(x).squeeze(1)

  def training_step(self, batch, batch_idx):
    x, target = batch
    target = target.float()
    logits = self.forward(x)
    loss = self.loss_fn(logits, target)
    return loss

  def configure_optimizers(self):
    return optim.Adam(self.parameters())
```

```
class MLP(LinearClassifier):
  def __init__(self, input_size, hidden):
    super().__init__(input_size)
    self.nn = nn.Sequential(
        nn.Linear(input_size, hidden),
        nn.ReLU(),
        nn.Linear(hidden, 1)
    )
```

Initial Network Design

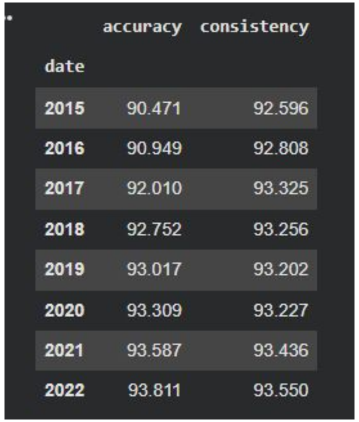
Training

- Trained with a batch size of 256

- **Binary Cross Entropy** was used with the **Adam** optimizer

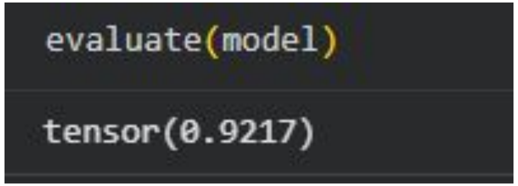
- The initial model achieved **~92% accuracy.**

    - This is already about as good as the accuracy can get, now we just need to make the model more confident!



date	accuracy	consistency
2015	90.471	92.596
2016	90.949	92.808
2017	92.010	93.325
2018	92.752	93.256
2019	93.017	93.202
2020	93.309	93.227
2021	93.587	93.436
2022	93.811	93.550

← Actual Umpire Accuracy Data



```
evaluate(model)

tensor(0.9217)
```

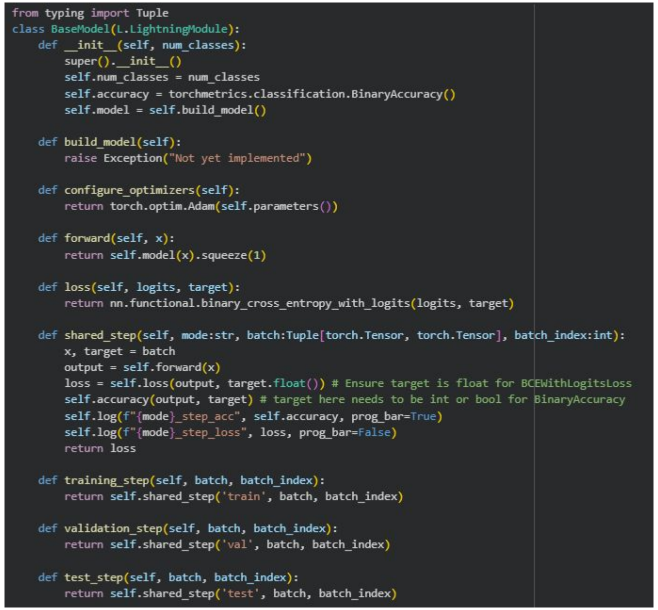
We got 92% accuracy for 2022, seems about right!

More Robust Network Design

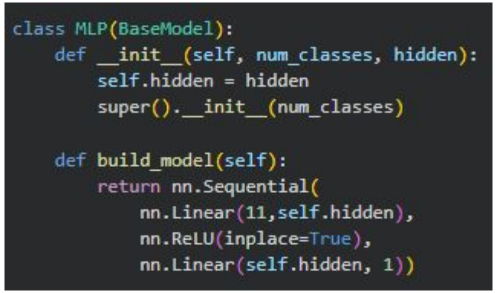
We decided to improve upon our previous model, and add some features we used in assignment 2:

- Lightning Logs
- Checkpoints
- Ability to show training + validation curve

We also decided to adjust the data loader, explicitly defining a training/validation split of **70:30**



```python
from typing import Tuple
class BaseModel(L.LightningModule):
    def __init__(self, num_classes):
        super().__init__()
        self.num_classes = num_classes
        self.accuracy = torchmetrics.classification.BinaryAccuracy()
        self.model = self.build_model()

    def build_model(self):
        raise Exception("Not yet implemented")

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters())

    def forward(self, x):
        return self.model(x).squeeze(1)

    def loss(self, logits, target):
        return nn.functional.binary_cross_entropy_with_logits(logits, target)

    def shared_step(self, mode:str, batch:Tuple[torch.Tensor, torch.Tensor], batch_index:int):
        x, target = batch
        output = self.forward(x)
        loss = self.loss(output, target.float()) # Ensure target is float for BCEWithLogitsLoss
        self.accuracy(output, target) # target here needs to be int or bool for BinaryAccuracy
        self.log(f"{mode}_step_acc", self.accuracy, prog_bar=True)
        self.log(f"{mode}_step_loss", loss, prog_bar=False)
        return loss

    def training_step(self, batch, batch_index):
        return self.shared_step('train', batch, batch_index)

    def validation_step(self, batch, batch_index):
        return self.shared_step('val', batch, batch_index)

    def test_step(self, batch, batch_index):
        return self.shared_step('test', batch, batch_index)
```

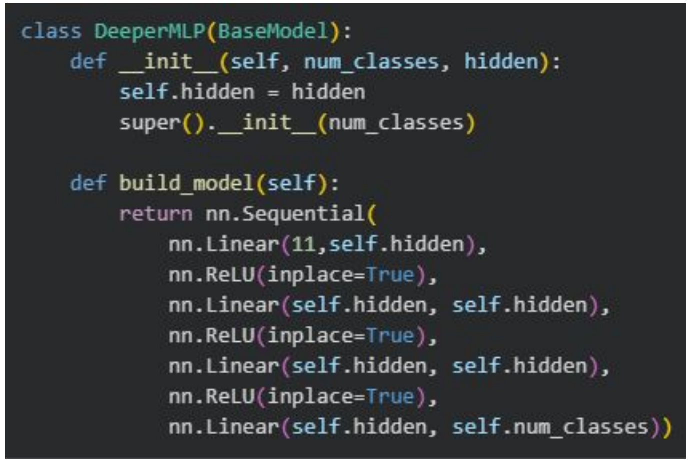


```python
class MLP(BaseModel):
    def __init__(self, num_classes, hidden):
        self.hidden = hidden
        super().__init__(num_classes)

    def build_model(self):
        return nn.Sequential(
            nn.Linear(11,self.hidden),
            nn.ReLU(inplace=True),
            nn.Linear(self.hidden, 1))
```

Further experiments with different network designs

Experiment 1: Increasing Depth

- Tested a new class, “DeeperMLP”
- Adding 2 more hidden layers, each with 64 neurons
- This resulted in a ~92% accuracy again, though log loss didn’t change much



```python
class DeeperMLP(BaseModel):
    def __init__(self, num_classes, hidden):
        self.hidden = hidden
        super().__init__(num_classes)

    def build_model(self):
        return nn.Sequential(
            nn.Linear(11,self.hidden),
            nn.ReLU(inplace=True),
            nn.Linear(self.hidden, self.hidden),
            nn.ReLU(inplace=True),
            nn.Linear(self.hidden, self.hidden),
            nn.ReLU(inplace=True),
            nn.Linear(self.hidden, self.num_classes))
```

Further experiments with different network designs

Experiment 2: Learning Rate Tuning:

- We then tested with a few different learning rates:
  - 0.01: Overcorrected and trained too fast
  - 0.0001: Too slow, resulting in a low final accuracy (<70%)
  - 0.001: Just right!

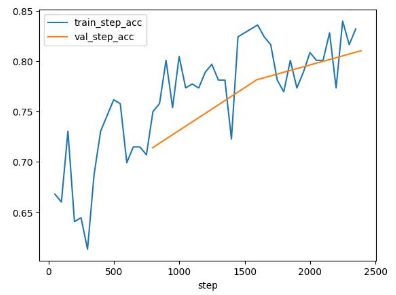
cred: https://sites.temple.edu/texttotalk/goldilocks/

Further experiments with different network designs

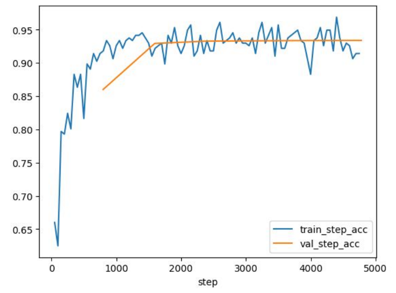
Experiment 3: Feature Scaling / Normalization

- We implemented the **StandardScaler** from Scikit-Learn to **normalize** the inputs.
- This prevents features with large values (such as release_speed) to **overpower** the smaller movement features
- This improved training speed significantly, and improved log loss as well!

Before:



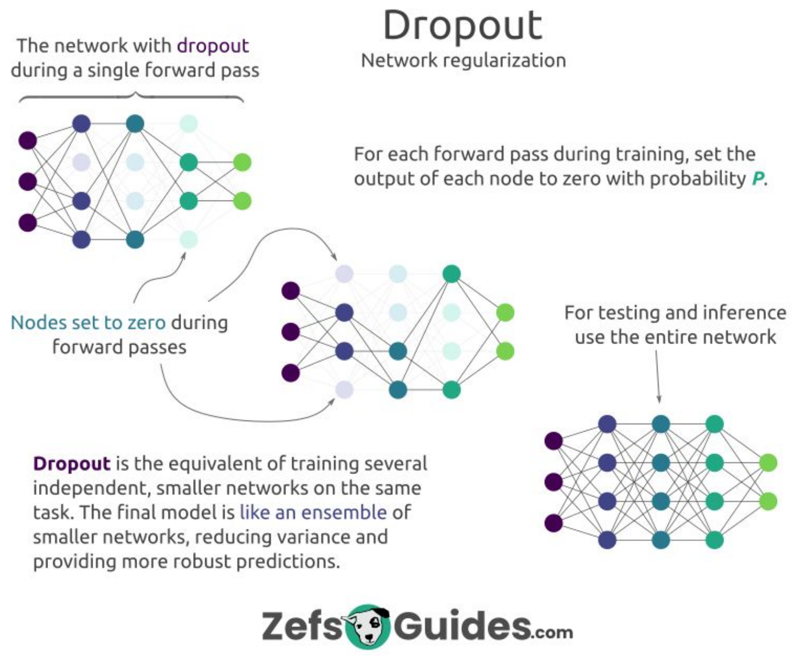
Now:



Further experiments with different network designs

Experiment 4: Dropout Regularization:

- We added a dropout rate of 0.2 to training, paired with a larger hidden layer size.
- This final adjustment aimed to lower log loss further while maintaining the ~92-94% accuracy ceiling.


Dropout
Network regularization
The network with dropout during a single forward pass
For each forward pass during training, set the output of each node to zero with probability P.
Nodes set to zero during forward passes
For testing and inference use the entire network
Dropout is the equivalent of training several independent, smaller networks on the same task. The final model is like an ensemble of smaller networks, reducing variance and providing more robust predictions.
ZefsGuides.com

Deployment

Gradio app deployed on Hugging Face

https://huggingface.co/spaces/ryan-don31/ML-Final-Project-DavidRyan

Results

- We picked a close example where the pitch should have been called a ball
- Link to the Gradio app: https://huggingface.co/spaces/ryan-don31/ML-Final-Project-DavidRyan
- Real pitch example: https://baseballsavant.mlb.com/sporty-videos?playId=2d5f1447-66e2-4c57-ade6-3f0247aac218

In this case, the umpire mistakenly called this pitch a strike, just as our model predicted!

It’s also worth noting that changing the game state affects the confidence significantly.