Project Summary

In this project, I will be using a neural network to predict the pitch type of MLB pitches thrown between 2015-2018, based on the advanced metrics caputred by the high quality Statcast cameras, such as the spin rate of the pitch, the movement in the vertical and horizontal directions, the angle of break on the pitch, and much more. I sourced this data from Kaggle (https://www.kaggle.com/datasets/pschale/mlb-pitch-data-20152018/discussion?sort=undefined). As an avid fan of baseball, I thought this question posed the perfect oppurtunity to delve into the world of deep learning and the PyTorch Deep Learning Library with a hands on project in an area of interest; clearly, the type of pitch thrown is heavily correlated with the movement charecteristics of the ball, and a neural network can definitelty capture this relationship.

Data Preparation

In [ ]:

```
## Imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
```

In [ ]:

```
#Read ind data. Data set from Kaggle https://www.kaggle.com/datasets/pschale/mlb-pitch-data-20152018/discussion?sort=undefined
data = pd.read_csv('pitches.csv')
```

In [ ]:

```
#Drop columns that are not viable predictors
irrel = ['type_confidence', 'ab_id', 'on_1b', 'on_2b', 'on_3b', 'zone', 'code', 'type', 'event_num',
                            'nasty', 'b_score', 'b_count', 's_count', 'outs','pitch_num']
for col in irrel:
    data = data.drop([col], axis = 1)

## Filter for Fastballs, Curveballs, Sliders, and Changeups
## These are the most common kinds of pitches, and most useful to be predicted
data = data[(data['pitch_type'].str.strip() == 'FF')  | (data['pitch_type'].str.strip() == 'CH') |
(data['pitch_type'].str.strip() == 'FT') | ((data['pitch_type'].str.strip() == 'CU')) | (data['pitch_type'].str.strip() == 'SL') ]

#2 kidns of fastballs (4-seam and 2-seam). Merge them into one "Fastball" as they are very similar
data['pitch_type'] = data['pitch_type'].replace({'FF': 'FB', 'FT': 'FB'})

#Drop NA values
```

```
data = data.dropna(how = 'any', axis = 0)
X = data.drop(['pitch_type'], axis = 1)
y = data['pitch_type']

#Encode Pitch Tybes with Label for easier Neural Network computations
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
```

In [ ]:
```
# Standardize values
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

In [ ]:
```
### Perform PCA
## Select top 5 most "influential" variables (variable w/ max value from each principle direction)
num_components = 5
pca = PCA(n_components=num_components)
comps = pca.fit(X_scaled)
best_vars = [np.argmax(comps.components_[num]) for num in range(num_components)]
X_ready = X_scaled[:, best_vars]
## Display variable names
var_names = [list(X.columns)[np.argmax(comps.components_[num])] for num in range(num_components)]
print(var_names)
```

```
['vy0', 'ax', 'x', 'pz', 'px']
```

Here, we see that the top 5 most influential variables for the 5 principal directions calculated in PCA are the initial velocity of the pitch in the direction of the hitter (vy0), acceleration in the x direction (ax), the break in the horizontal direction (x) and the location of the horizontal (px) and vertical (pz) locations of the pitch with respect to the hitter and home plate. We can use this as a logic check, as all of these values are intuitive. Fastballs have high velocity and can be identified by their vy0, breaking pitches like sliders and curveballs that move more will have higher ax and x values, and they are also more likely to be located on the edge of the plate and low in the strike zone than a fastball would be. Therefore, we will proceed to use these 5 variables.

Create Neural Network

In [ ]:
```
## Define NN architecture
## This one will have 2 hidden layers, one with 8 nodes and one with 4
## Input dimension is 5, for the 5 predictor variabkes
## Output dimension is 4, for the 4 distinct pitch types
input_size = 5
hidden_size1 = 8
hidden_size2 = 4
output_size = 4

class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(SimpleNN, self).__init__()
        # Define the layers
        self.fc1 = nn.Linear(input_size, hidden_size1)  # Input to first hidden layer
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)  # First hidden layer to second hidden layer
        self.fc3 = nn.Linear(hidden_size2, output_size)  # Second hidden layer to output layer
        self.relu = nn.ReLU()  # ReLU activation function

    def forward(self, x):
        x = self.relu(self.fc1(x))  # Pass through first hidden layer
        x = self.relu(self.fc2(x))  # Pass through second hidden layer
        x = self.fc3(x)  # Output layer (no activation function, handled by loss function)
        return x
```

In [ ]:
```
# Create instance of the model
model = SimpleNN(input_size, hidden_size1, hidden_size2, output_size)

# Cross entropy loss functiom - differentiable so easy for gradient descent
criterion = nn.CrossEntropyLoss()

#optimizing agent
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [ ]:
```
# Partition to train and test sets
x_train, x_test, y_train, y_test = train_test_split(X_ready, y_encoded, test_size=0.2, random_state=13)

# Convert data to PyTorch tensors
x_train_tensor = torch.tensor(x_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
x_test_tensor = torch.tensor(x_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

# Create TensorDataset and DataLoader for training
train_dataset = TensorDataset(x_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
```

Train Network

In [ ]:
```
# Initialize 20 epochs
num_epochs = 20
for epoch in range(num_epochs):
    for batch_X, batch_y in train_loader:

        ## Zero the parameter gradients
        optimizer.zero_grad()

        ## Go forward through the network
        outputs = model(batch_X)
        ## Compute the loss for this step
        loss = criterion(outputs, batch_y)

        ## Backpropogate and tweak parameters
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

print("Training complete")
```

```
Epoch [1/20], Loss: 0.3353
Epoch [2/20], Loss: 0.4264
Epoch [3/20], Loss: 0.4057
Epoch [4/20], Loss: 0.2198
Epoch [5/20], Loss: 0.0868
Epoch [6/20], Loss: 0.5031
Epoch [7/20], Loss: 0.1398
Epoch [8/20], Loss: 0.4426
Epoch [9/20], Loss: 0.3157
Epoch [10/20], Loss: 0.6233
Epoch [11/20], Loss: 0.2960
Epoch [12/20], Loss: 0.4582
Epoch [13/20], Loss: 0.3149
Epoch [14/20], Loss: 0.3772
Epoch [15/20], Loss: 0.2189
Epoch [16/20], Loss: 0.3709
Epoch [17/20], Loss: 0.2900
Epoch [18/20], Loss: 0.3822
Epoch [19/20], Loss: 0.7154
Epoch [20/20], Loss: 0.8462
Training complete
```

Test Network

In [ ]:
```
# Function to evaluate accuracy
def evaluate_accuracy(model, data_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():  # Disable gradient calculation
        for batch_X, batch_y in data_loader:
            outputs = model(batch_X)
            # Get the index of the max log-probability
            _, predicted = torch.max(outputs, 1)
            total += batch_y.size(0)
            ##Compute correct predictions
            correct += (predicted == batch_y).sum().item()
    accuracy = correct / total
    return accuracy

## Create DataLoader object for the test dataset
test_dataset = TensorDataset(x_test_tensor, y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)

## Evaluate test accuracy
test_accuracy = evaluate_accuracy(model, test_loader)
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')
```

```
Test Accuracy: 86.28%
```