Tennis Shot Classification: Using Machine Learning to Classify Types of Tennis Shots in Video Data

Irene Agusti, Utsah Kalra, Sam Rudenberg

May 17, 2024

Abstract

We created machine learning models to predict the type of tennis shot in a given frame of a video. Video data was split into sequential frames, which were manually labeled by the type of shot discernible by the human eye. We created a CNN model, and attempted to create a 3DCNN model, and evaluated them on training and testing data. Our results found that our CNN model reached an accuracy of 17% on the training set, and 19% on the test set.

Intro

In this project, we use a data set containing snapshots from a tennis match to classify the different types of tennis shots being hit in that snapshot. There are many machine learning algorithms that one can use to do this. In our project, we choose to focus on using Convolutional Neural Networks to perform this classification task. While CNN's are computationally expensive, they are well-suited to data that requires identification and extraction of spatially dependent hidden features (slides).

Methods

The data used for the project comes from Github user yastrebksy. This user has a repository for TennisProject, which is described as Tennis analysis using deep learning and machine learning. A subcomponent of this user's project, TrackNet is used to take video data as input, and output frame by frame information with file names, visibility, xy-coordinates, and status. File names are individually labeled frames. Visibility describes how visible a tennis ball is within a given frame; 0 for ball not in image, 1 for ball can be easily identified, 2 for in frame, but not easily identified. xy-coordinates are provided for the location of the ball in the frame (where 0,0 is the bottom left corner). Status has three options: hitting, flying, or bouncing.

We downloaded images and outputs from TrackNet for Games 1 and Games 8. The images we downloaded are pre-cut, frame by frame pieces of a 30-fps video clip. After downloading all the images, we wrote code to rename them, appending the game and clip number, along with the sequence number of the image where 0 is the first frame of the video, and the largest number is the last frame of the video. We added a column to the TrackNet output called shot, with the following classifications: 1 - forehand, 2 - backhand, 3 - forehand slice, 4 - backhand slice, 5 - serve, 6 - forehand volley, 7 - backhand volley, 8 - overhead, 9 - standby. After all data was cleaned and labeled, we moved on to the model. Below is a breakdown of the frames for each clip in game 1 as well as game 8. We renamed the images as seen in the sequenced columns so that all images could be in the same folder, while having a way of being differentiated from the other clips. This was a replacement for the earlier game.clip.frame appendage that we used for our 3DCNN model.

Game 1, Clip:	Sequenced	Total Frames
1	1-207	207
2	208-337	130
3	338-400	63
4	401-508	108
5	509-668	160
6	669-746	78
7	747-1198	452
8	1199-1313	115
9	1314-1422	109
10	1423-1552	130
11	1553-1678	126
12	1679-1931	253

Game 8, Clip:	Sequenced	Total
1	1932-2049	118
2	2050-2144	95
3	2145-2527	383
4	2528-2660	133
5	2661-2739	79
6	2740-2805	66
7	2806-3227	422
8	3228-3546	319
9	3547-3784	238

In the context of our model, our predictors/features are the images and the targets/classification are the shot type labels, which we created ourselves. Our image data size was started at [720, 1280, 3].

We began to create our model. Our first model was a CNN image classification model with five layers: Convolutional, ReLU, Max Pool, and two linear layers. The first layer was the convolutional layer, which consisted of 3 RGB inputs, 16 features maps, a kernel size of 5, stride of 1, and padding of 2. The next layer was the ReLU layer, which acted as an activation layer. This was followed by a Max Pool layer with a kernel of 2, and stride of 2, which reduced the size of the feature maps by taking the max value. Lastly, we have two linear layers which reduce the inputs from 134400 to 128, to 9 outputs which serve as our classifications. This was presented in class.

When we first ran our training data on the model, we encountered issues with RAM. Our image size was too large, and was causing our computers to run out of memory. To counteract this, we decided to reduce our image size. We used interpolation to reduce our image size to 140x240. By reducing the image this much, there is a risk of losing quality, but it was necessary to make sure our computer's had enough RAM to run the model. After running the model on the training data, we evaluated the model using cost curves and classification accuracy. Our initial accuracy was 20%.

After our class presentation, we discovered that the data set we had downloaded from the Github project had incomplete games, which meant our results were invalid. We had originally used clips from game 9, rather than the clips from game 8 which are seen above, but this meant we had to evaluate our model.

As we re-evaluated our model, it was even more demanding in terms of RAM. We converted our images to grayscale, and simplified our model. We experimented with different batch sizes, learning rates, and framework layers to maximize accuracy while still being small enough for our computer to run. Our network became a series of three linear layers with ReLU activation between them, reducing from 33600 inputs, to 8192, to 512, to 10. We used 5 epochs, a learning rate of 0.01, and a batch size of 100.

While a CNN model allows us to exploit the spatially dependent features in image data, it does not allow us to exploit the sequential nature of data. Here in our project, the data set was created from video data and has an important sequential aspect to it. Using Recurrent Neural

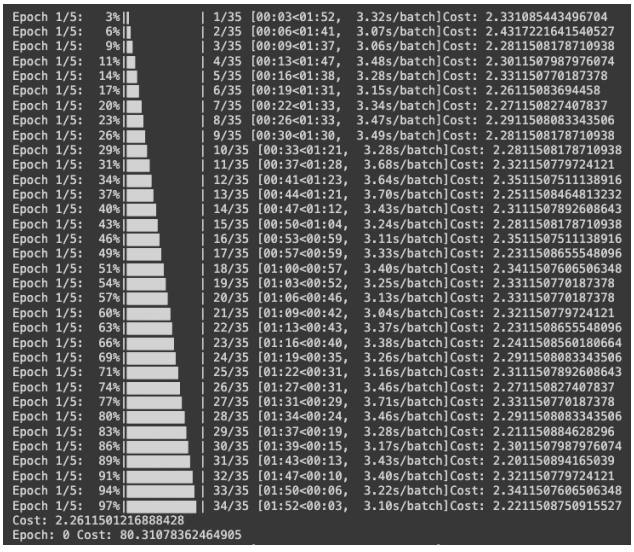
Networks allows us to take into account the sequential aspect of our data set. The mechanism through which rnn's performs sequential learning is the "hidden state." This "hidden state" maintains information about the previous frame, and uses both the frames features itself, combined with the information from the previous frame to predict the next outcome.

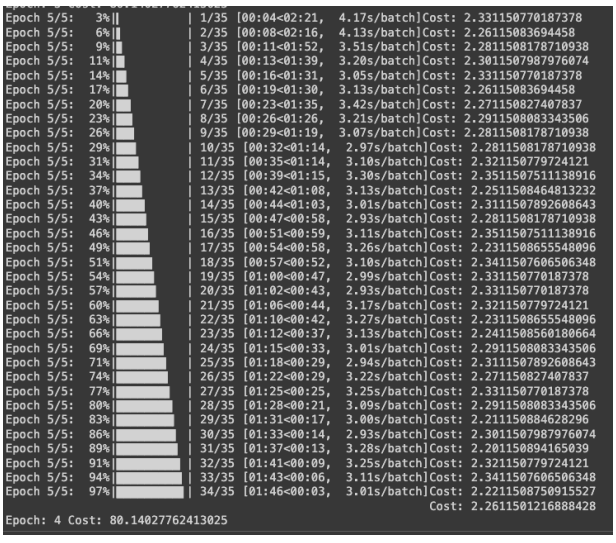
More specifically, for our second comparative model using recurrent neural networks, we use a LSTM (Long Short Term Memory) + CNN model. The LSTM+ CNN model handles the sequential learning and relative feature learning aspects separately. CNN handles the spatial feature extraction while the LSTM focuses on temporal relationships. This separation can make it easier to tune to each part of the model.

Another way to take into account the sequential nature of data is to perform a 3D CNN model. A 3D CNN model, treated the sequence of frames as a third layer dimension of the data instead of dealing with it separately as in the LSTM+CNN model. For this model, we have a convolutional layer with 16 feature maps, a kernel size of 3, a stride of 3 and padding of 1; a max pooling layer with kernel 2, and stride of 2. Finally, we have 2 fully connected layers, the first one takes an input of 134,400 and compresses it to output 128 features; the second connected layer reduces it to the final 10 outputs that serve as our classifications.

Results

After 1 epoch, our model could do no further learning due to the simplicity of our model. Our final CNN model made correct predictions on the training set 17% of the time, and 19% of the time on the test set. Due to the complexities and time restrictions, we were unable to run our other models. Below are epochs, epoch images from running the model.





```
# Initialize objects for counting correct/total
correct = 0
```

Run cell (⌘/Ctrl+Enter)
cell executed since last change

executed by Sam Rudenberg
3:55 PM (42 minutes ago)
executed in 48.781s

```
                                                  in the subsequent steps (since we're not using these data for training)

                                                esc='Evaluation', unit='batch', leave=False):

            # pass each batch into the network
            outputs = net(images)

            # the class with the maximum score is what will be the predicted class
            _, predicted = torch.max(outputs.data, 1)

            # add size of the current batch
            total += labels.size(0)

            # add the number of correct predictions in the current batch
            correct += (predicted == labels).sum().item()

    # Calculate and print the proportion correct
    print(correct/total)
```

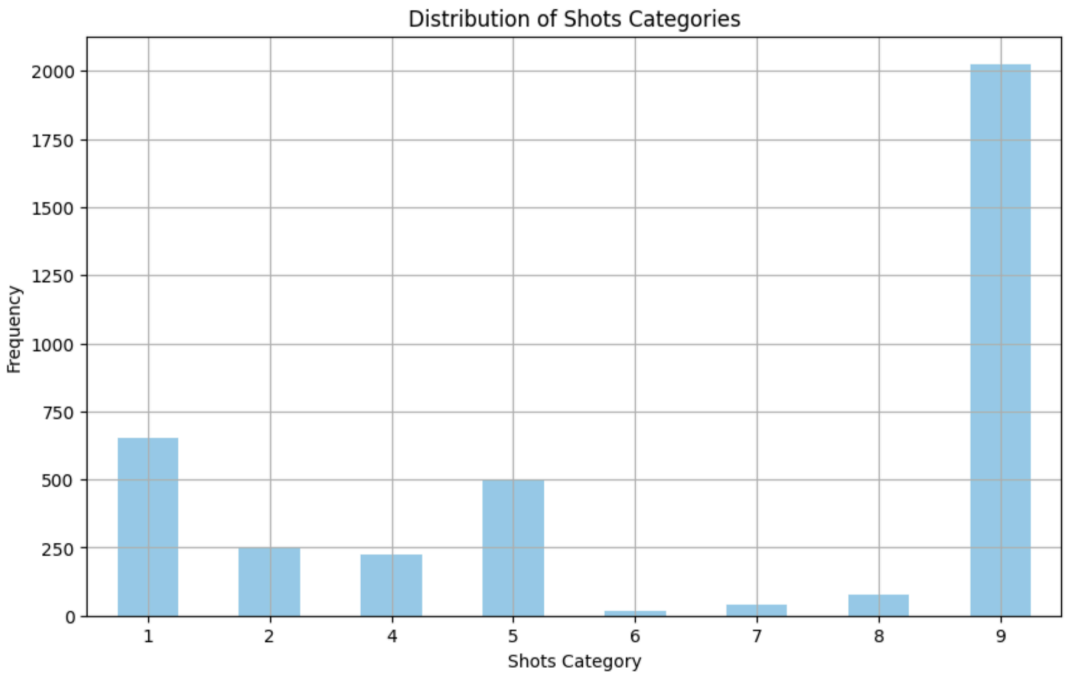
```
0.17063142437591777
```

```
# Convert Pandas Series to NumPy array
test_y_array = test_y.to_numpy()

# Convert NumPy array to PyTorch tensor
test_y_tensor = torch.Tensor(test_y_array)

### Convert to numpy array then reshape
test_unflattened = test_X.reshape(len(test_y), 240, 140, 1)

## Convert test images into a tensor
test_tensor = torch.from_numpy(test_unflattened)

## Combine X and y tensors into a TensorDataset and DataLoader
test_loader = DataLoader(TensorDataset(test_tensor.type(torch.FloatTensor), test_y_tensor.type(torch.LongTensor)), batch_size=bsize)

## Repeat evaluation loop suing the test data
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
print(correct/total)
```

```
0.18997361477572558
```

Above is the accuracy on the training and test sets.

Taking a deeper look at our data, there are class imbalances. The majority of frames in each clip are standby, and each clip always begins with a serve, however, the other shots appear considerably less, if at all. If we were to repeat this project, we would attempt to find videos to equalize our categories. Below is a distribution of the types of classifications across all clips.



Discussion

The models we embarked on doing were definitely very ambitious and required access to a more powerful pc that could be running the code in the background. Ideally we should have found a suitable pre-trained model to help us "deload" some of the pressure in the RAM. The big challenge involved taking into account the temporal aspect of the data since the images where labeled to be treated together. As expected, running a model that did not take time into account does not have an optimal accuracy, randomly guessing "standby" would provide us with 50% accuracy.

References

Miller, R. E. (n.d.). Machine Learning - Supervised Learning: Introduction to Regression [Webpage]. Retrieved from https://remiller1450.github.io/MLs24.html