Capstone Report

Machine Learning Engineer Nanodegree

Logan Spears
August 7, 2017

Definition

Project Overview

Video classification is gaining attention as image classification becomes more and more accurate. Datasets such as Youtube 8M and UCF101 are enabling the development of better models for classifying videos in the same way that ImageNet and others did for images. While reviewing the UCF101 dataset, I noticed that "tennis swing" was a activity in the dataset. As a tennis player and fan myself, I began to imagine an AI that could function as a line judge, score keeper, or game statistician. In this project I have chosen to implement a critical step in understanding abstract concepts of a tennis match: shot type classification.

Problem Statement

Tennis is a dynamic game with multiple shot types: forehand, backhand, volley, serve, etc. Even an inexperienced tennis player or fan can distinguish between these shots. The goal of this project is to create an AI that replicates this functionality by completing the following tasks:

1. Acquire and preprocess videos of tennis practice and matches
2. Label clips within the video by shot type
3. Train a model on the video input features and labels
4. Run predictions with the model until it performs with sufficient accuracy

Metrics

Multi-class classification problems typically use accuracy as an evaluation metric. Accuracy is defined by the following equation:

$$accuracy = \frac{tp + tn}{n}$$

True positive, *tp*, represents the number of correct predictions when the result was positive out of *n* inputs. True negative, *tn*, represents the number of correct predictions when the result was negative. The result being positive or negative is essentially a binary multi-class classification problem. A generalized version of accuracy is defined by the following equation:

$$accuracy = \frac{tp_1 + tp_2 + tp_3 \ldots tp_i}{n}$$

In this modified equation, the sum of true positives from all classes is divided by the number of inputs. While accuracy is useful as a primary metric of performance, it doesn't give insight into which classes the model is underperforming on. A confusion matrix can supplement accuracy and give more detail about model performance and show which classes, or labels, the model is struggling with.

Figure 1: Example Confusion Matrix

	Cat Predicted	Dog Predicted
Cat Actual	7	3
Dog Actual	3	3

A confusion matrix visualizes two additional metrics alongside true positives and true negatives: false positives, *fp,* and false negatives, *fn*. False positives are predictions of a particular label when it is actually another value. In Figure 1 (above) the number of false positives for Cat are counted in the [Dog Actual, Cat Predicted] element. The number of false negatives for Cat are counted in the [Cat Actual, Dog Predicted] element. Using all squares from the matrix we can calculate that the prediction accuracies of the labels 'cats' and 'dogs' in this example are 70% and 30%, respectively.

Analysis

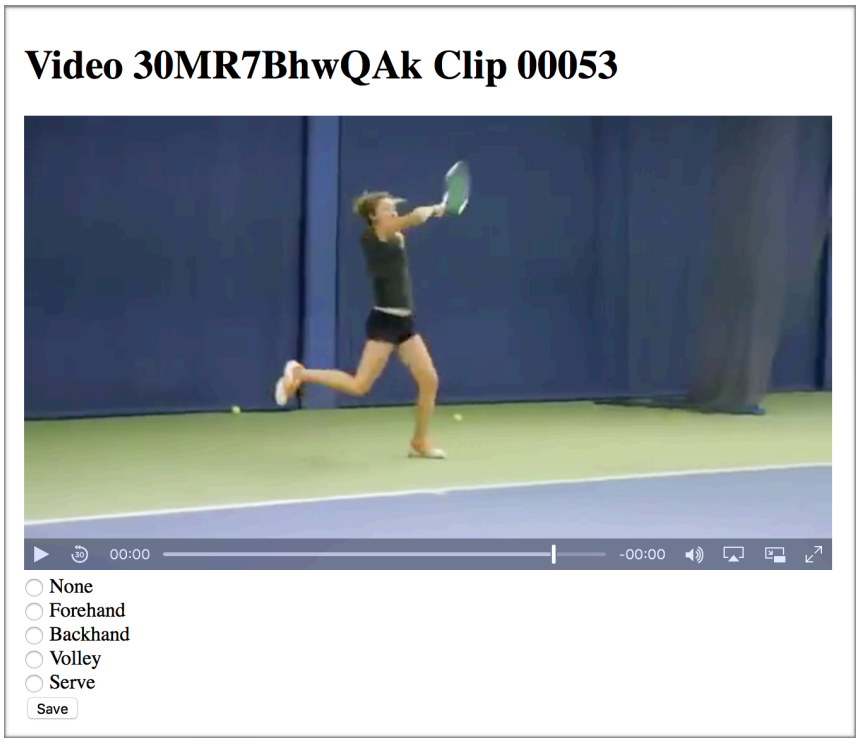
Data Exploration

The datasource for this project was YouTube, which was selected for its large video catalog and its search filters for Creative Commons licenses. Videos that contained scenes of people playing tennis in a professional or causal setting were searched for and downloaded. Each video was downloaded in 480p (480 x 640 resolution, acc 44100 Hz audio) in the MP4 container format.

The final training set was derived from nine videos. Each video was segmented into one-second clips that were subsequently labeled. Figure 2 illustrates the web form used to speed up the labeling process. It presents an html page that autoplays the one-second clip and presents a form for selecting the shot type. Selecting the "None" label categorizes the clip as not usable. Of the 5090 clips labeled, only 926 contained a clear shot usable for training. Reasons for labeling a clip as "None" included: (a) there being no shot in the clip; (b) there being multiple shots in the clip; (c) a shot being on the edge of the clip; (d) a player going off camera during the clip; and (e) a player being too far away from the camera.

Shot types were intentionally limited to forehand, backhand, volley, and serve to simplify the labeling and training process. Overheads, forehand volleys, backhand volleys, backhand slices, approach shots, half volleys, tweeners (between-the-legs shots which are always a crowd favorite), and many other shot types and subtypes could have been chosen for labeling, but were either too infrequent or inconvenient to label.

Figure 2: Labeling Web Form



Exploratory Visualization

Due to natural differences in video length and composition, each video produced a different number of training examples and percentages of shot types. Figures 3 and 4 below show the breakdown of shot type by video and overall shot percentages. Because college recruitment videos typically showcase many or all of a players' skills, these videos were specifically targeted to achieve more equal proportions. However, some shot types were just more common than others. For instance, forehands were more than twice as common as volleys in the dataset. To prevent the model from ignoring uncommon shots like volleys, the confusion matrix was added to the final accuracy measurement to prove the model wasn't just selecting the more common shots.

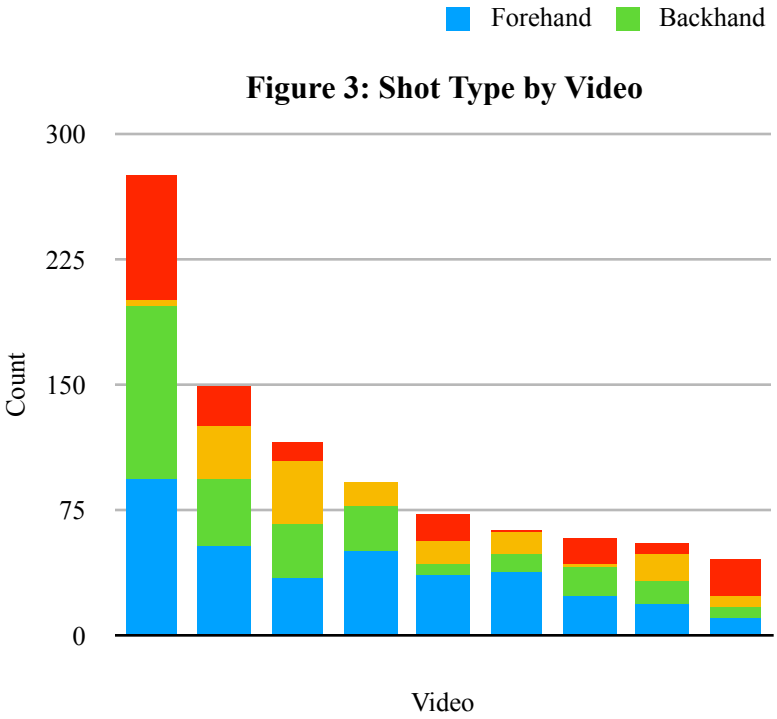


Figure 3: Shot Type by Video

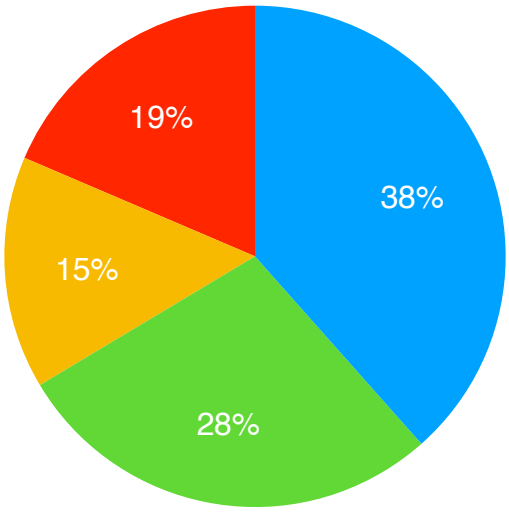


Figure 4: Shot Type Percentages

Algorithms and Techniques

Shot classification is modeled as a multi-class classification problem using video data as input. The model utilizes Google's InceptionV3 pre-trained Convolutional Neural Network (CNN), Long Short Term Memory (LSTM) based Recurrent Neural Network (RNN), and standard dense Neural Network (NN) layers to create a Deep Neural Network (DNN) capable of understanding and classifying what type of shot is in a given video clip. InceptionV3 was chosen because it is provided by the Keras framework.

CNNs are the industry standard for analyzing visual imagery because of convolutional and pooling layers that are both efficient and translation tolerant. CNNs trained on large datasets, such as ImageNet, learn weights that form layers which identify patterns of ascending complexity. These weights can be reused with the convolution and pooling layers to grant new models the same visual comprehension

RNNs are designed to learn from ordered data such as natural language or time series data. Because a video is a series of images over time, it is a good candidate for an RNN based model (Harvey). LSTMs, a

type of RNN, retain a memory of both recent and distant features of the input and so generalize well to inputs of different lengths.

"Dense" or "Fully Connected" layers represent the original form of NNs in which all nodes are connected. Dense layers typically are added to an architecture after convolutional or recurrent layers in order to connect them to a sigmoid or softmax activated output layer, which enables binary or multi-class classification. Dense layers can also add complexity to the model if the features are relatively independent.

Benchmark

Forehand, the most common shot type, was the label for 38% of the training examples (see Figure 4 above). A model that predicted forehand on every example could therefore achieve an overall accuracy of 38%. To prove a model has learned from the training set, it must surpass this value by a statistically significant amount. Because of the time investment required to label data, this project aimed to beat a benchmark of 60% overall accuracy.

Methodology

Data Preprocessing

Clips

The initial preprocessing step slices full length videos into one-second clips. ffmpeg, a video editing tool, has a command that takes a video path, start time in seconds, and length in seconds to produce a clip. This command, shown below, is run for every second of every video to produce the clip dataset for future preprocessing stages.

```
$ ffmpeg -i vid1.mp4 -ss 0 -t 1 vid1/00000.mp4
```

Frames

After the clip dataset is created, individual JPEG frames are extracted from the clip MP4. The ffmpeg command below takes the clip created in the previous section and creates seven frames (five specified by -r plus a beginning and ending frame) from it.

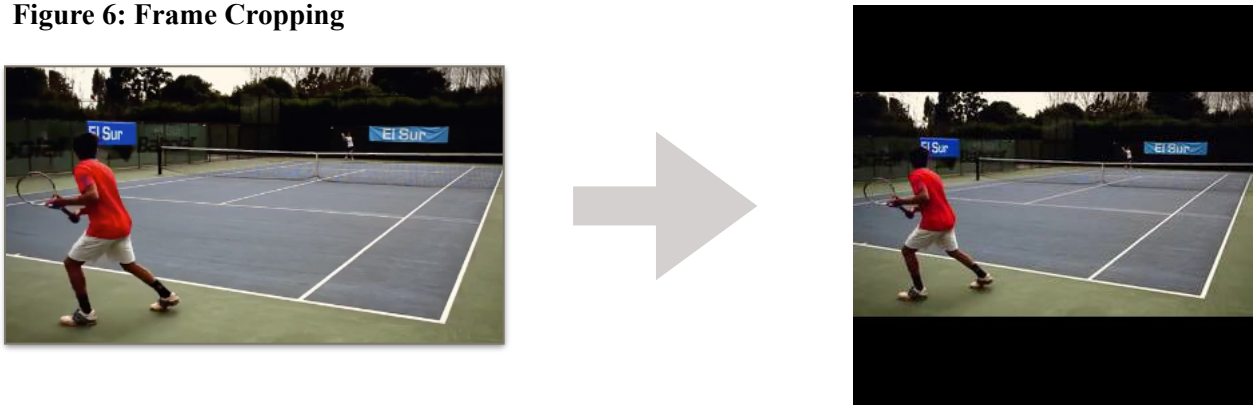
```
$ ffmpeg -i vid1/00000.mp4 -r 5.0 vid1/00000/%5d.jpg
```

After each JPEG frame is created, it is cropped to 299x299 pixels in order to be used by the feature extraction preprocessing step. ImageMagick, an image manipulation tool, exposes the convert command which enables many cropping tasks. The first command shown below resizes the image to fit within a 299x299 bounding box while preserving the aspect ratio of the original image. The second command takes the output of the first command and centers it within a 299x299 image and blacks out unused pixels. The result is a 299x299 image that preserves the original images aspect ratio and can be used by the feature extraction step (see Figure 6).

```
$ convert vid1/00000/00000.jpg -resize 299x299 vid1/00000/00000.jpg


$ convert vid1/00000/00000.jpg -gravity center -background black -
extent 299x299 vid1/00000/00000.jpg
```

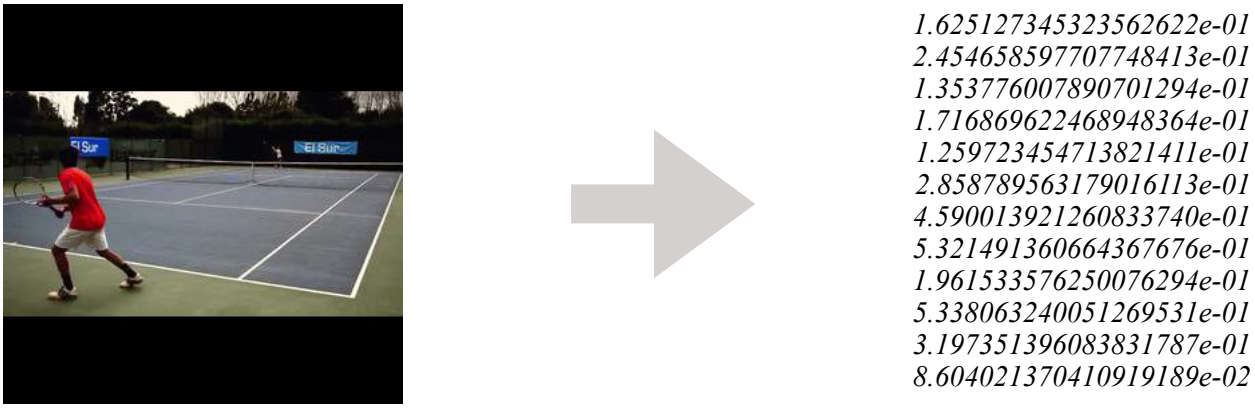
Figure 6: Frame Cropping



Features

Once the JPEG frames are resized to 299x299, they are ready to be used as input for InceptionV3's feature prediction. The InceptionV3 model, with its top dense and output layers removed, consists of convolutional and pooling layers that generate a set of features that can be used by other models. These features, in the form of a 2048x1 numpy array, are extracted from every frame, compressed, and saved to disk. Figure 7 visualizes this process and provides a few values from the output array.

Figure 7: Feature Extraction



1.625127345323562622e-01
2.454658597707748413e-01
1.353776007890701294e-01
1.716869622468948364e-01
1.259723454713821411e-01
2.858789563179016113e-01
4.590013921260833740e-01
5.321491360664367676e-01
1.961533576250076294e-01
5.338063240051269531e-01
3.197351396083831787e-01
8.604021370410919189e-02

Implementation

Labeler

Due to the lack of a preexisting dataset, one had to be created. The initial spreadsheet labeling process was painfully slow and was replaced with a simple webpage. The webpage presents a form with a HTML5 video tag that autoplays a clip and radio buttons for the labels. A local web server, written in the Go programming language, renders the html template and the static video files. Here are the steps of the labeling process:

1. cd into the "labelmaker" directory
2. Run the command "make run" in your terminal
3. Go to http://localhost:9090/view/:video_id/:clip_id where :video_id is the video name and clip id is the second index of the clip within the video
4. View the clip that autoplays
5. Select from the options: None, Forehand, Backhand, Volley, and Serve
6. Press "Save" to submit the form which has the following side effects:
    1. The label is paired with the video and clip ids and inserted into the sqlite database
    2. The browser redirects to the next clip (next second) and restarts at step four
7. Run "make csv" after labeling is complete. This command outputs a csv file usable by the classifier. Table 2 show an example of the output.

Figure 8: Labelmaker CSV Output

video_id	clip_id	label
HQGX7sfNEEU	10	3
HQGX7sfNEEU	11	1
HQGX7sfNEEU	12	0
HQGX7sfNEEU	14	1

Classifier

Three frameworks were considered for building the shot type classifier: Tensorflow, TFLearn, and Keras. Keras was eventually selected because incorporating InceptionV3 proved the deciding factor. This project's implementation and InceptionV3 preprocessing steps were inspired from this repo which also used Keras: https://github.com/harvitronix/five-video-classification-methods. The implementation steps are as follows:

1. Preprocessing
    1. Generate one second MP4 clips from source video
    2. Generate JPEG frames from clips
    3. Generate feature vectors from frames using the InceptionV3's convolutional and pooling layers
    4. Save all files (even intermediaries) for training and debugging

2. Label Creation
    1. Generate labels from labelmaker csv output
    2. Shuffle labels

3. Divide labels into training, validation, and test sets at a ratio of 3:1:1

3. Compile Model

    1. Define model with the following layers
        1. LSTM layer that can input the seven feature vectors created per clip in preprocessing
        2. Additional LSTM layer
        3. Flatten layer to transform LSTM layer output into a vector
        4. Dense layer with relu activation
        5. 50% Dropout layer
        6. Dense layer with relu activation
        7. 50% Dropout layer
        8. Dense output layer with softmax activation and four outputs representing forehand, backhand, volley and serve.

    2. Compile model with the following properties
        1. “categorical_crossentropy” loss function designed to work with softmax output layer which is similar to mean squared error but optimized for multi-class classification
        2. “adam” optimizer which utilizes momentum to dynamically change the learning rate
        3. “accuracy” add overall accuracy to the training output

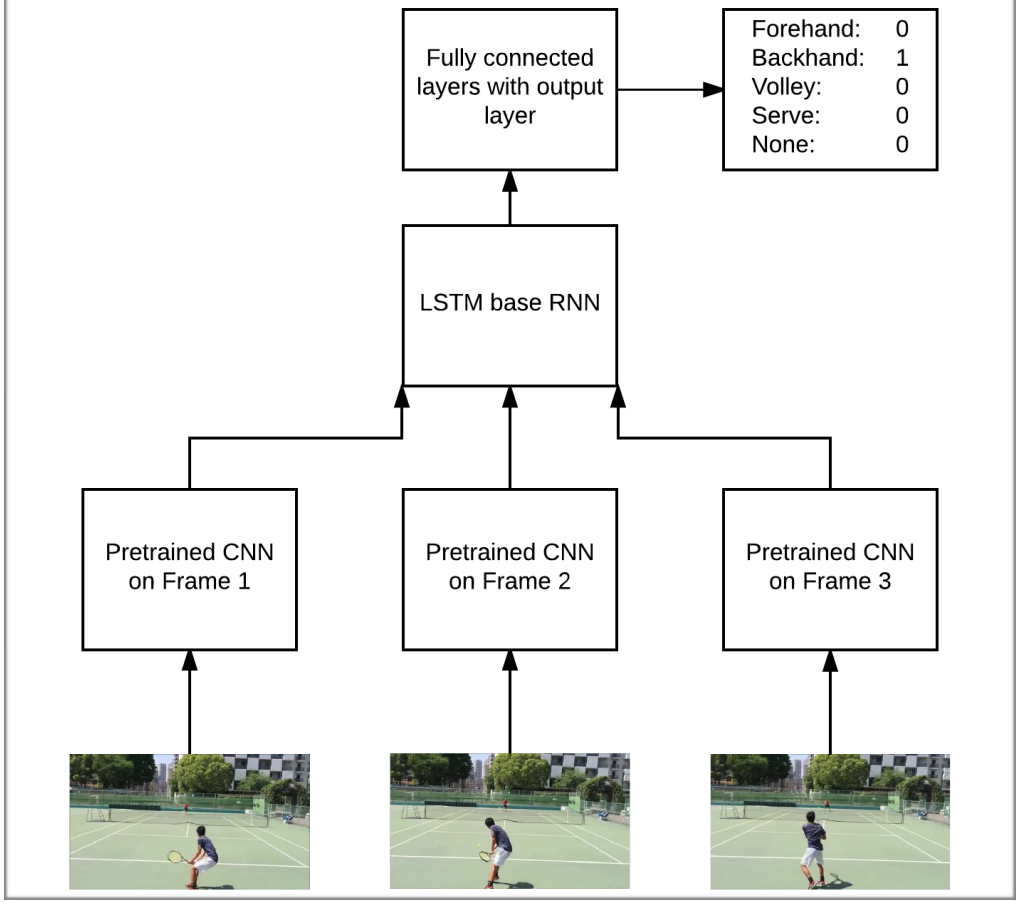
    3. Match Labels
        1. Clips are organized into inputs and labels
            1. Load and decompress saved feature vectors from disk
            2. Append each feature vector to others from clips creating a (7, 2048)
            3. One-hot encode the label (3 translates to [0,0,0,1])
            4. Add the (7, 2048) array to X at the same index as the one hot encoded label is added to Y
        2. Inputs and labels are created for training and validation datasets

    4. Training
        1. Instantiate a Tensorboard callback to view fitting results
        2. Train the model logging loss, accuracy, validation loss, and validation accuracy
        3. Save the model using the current timestamp to distinguish between runs

    5. Prediction
        1. Match the labels for testing dataset
        2. Load a saved model from disk
        3. Run the model prediction function using the testing labels
        4. Normalize the prediction output by reversing the softmax output into a label prediction
        5. Output the overall test accuracy
        6. Output the confusion matrix

Figure 9 illustrates the complete architecture.

Figure 9: Network Architecture



Refinement

Although the model stayed true to the initial design, the implementation changed significantly. The initial plan was to use InceptionV3's layers for each frame and connect them to LSTM and dense layers. After reading the Keras documentation, I discovered the merge function which takes multiple layers and combines their outputs into a single input for later layers. Merging the InceptionV3 models was problematic, however. I ran into a layer name conflict that results from some of the InceptionV3 layers having hard coded names. I filed a github issue (https://github.com/fchollet/keras/issues/7412), but wasn't terribly happy with the solution suggested. At the time the implementation only used three frames per video so I attempted to use three different pre-trained models and avoid the issue all together. While I eventually got a model to compile with this approach, it was very slow to train, didn't learn, and felt hacky.

I decided to search for a better solution. After google-ing and rereading documentation, I took another look at the video classification blog post I listed in my proposal's work cited page. The author of the blog post linked to a GitHub repository which contained his fully coded Keras implementation. His solution sidestepped the multiple InceptionV3 model problem by preprocessing the frames into feature vectors and saving them to disk. I recreated his methodology in my preprocessing steps and adapted the model to take feature vectors as input. After reworking every part of my code for this change, I reran the model

and … it still didn't learn, but I was actually amazed by the result. Instead of loading all the images from disk, processing them for InceptionV3, predicting the features, and feeding the results to the LSTM layers, the feature vectors were loaded from disk and fed in directly. Cacheing the feature vectors dropped the training time of the model from hours to a few minutes and in turn enabled rapid experimentation. After a few iterations of reviewing the loss and validation loss values, it became clear that my model wasn't complicated enough to represent the input. Adding an additional LSTM layer and retraining the model did the trick and it started learning! I tweaked the model by increasing the batch size and added epochs until I was happy with the results.

Results

Model Evaluation and Validation

Candidate models were evaluated using 555 training and 185 validation examples. Training metrics were logged after the completion of each Epoch. The values were also written to Tensorboard to better visualize each model's learning properties. The primary metrics of interest were loss and accuracy.

Loss represents the total error of the loss function for each epoch's batch. Figure 10 shows a slow and steady decrease of the training loss against a smaller more erratic decrease in the validation loss. The erratic behavior isn't surprising given the small size of the validation dataset. An important observation is the gradual divergence between the training and validation loss as the number of epochs increases. This phenomenon indicates overfitting.

Accuracy, which isn't actually used for training, represents the correct percentage of classifications for the epoch's batch. Figure 11 shows a gradual increase in test accuracy and a similar, albeit more volatile, pattern for the validation accuracy. Accuracy, which functions as a less precise proxy for loss, doesn't show detectable signs of overfitting.

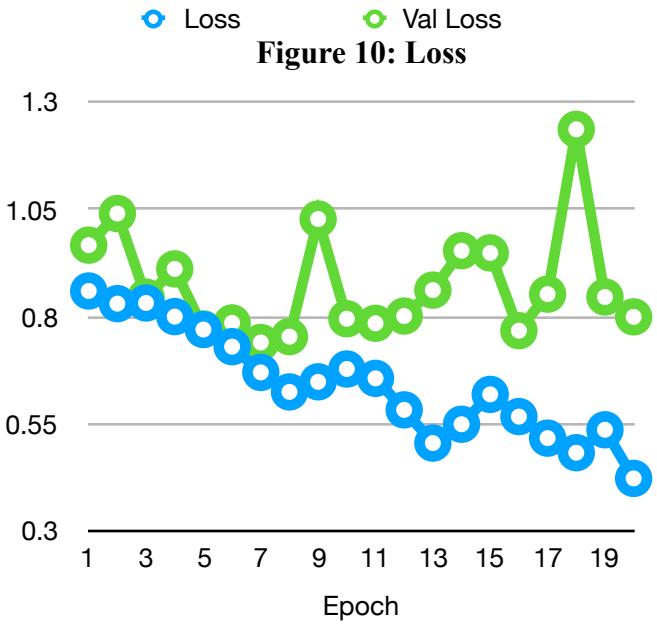


Figure 10: Loss

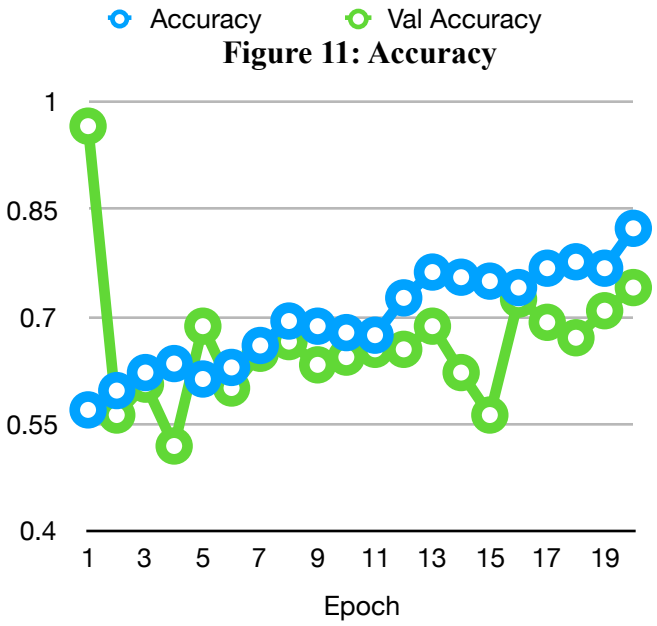


Figure 11: Accuracy

Justification

The model achieved an overall accuracy of 67.2% on the unseen testing dataset of 186 examples. This accuracy rating was higher than the initial benchmark of 60.0%, defined as a statistically significant result higher than a model could achieve without learning.

The confusion matrix, shown in Figure 12, showcases the relative strengths and weakness of the model in regards to particular shot types. The percentage of correct classifications for each shot type are: forehands 73.2%, backhands 31.3%, volleys 77.8%, and serves 92.5%. With the exception of backhands, the model did a respectable job of classifying each of the different shot types. The model was too aggressive about predicting forehands in general and especially for backhands. One possible explanation is that backhands were hard to distinguish for the model and it went with the shot that had the highest random chance of being correct. Even considering the backhand errors, the model is clearly learning. Considering the limited amount of training, validation, and testing examples, I was surprised to arrive at this good of a result.

Figure 12: Confusion Matrix

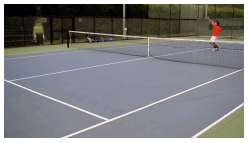
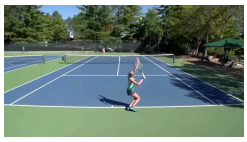
	Predicted Forehand	Predicted Backhand	Predicted Volley	Predicted Serve
Actual Forehand	52	12	1	6
Actual Backhand	29	15	0	4
Actual Volley	4	0	21	2
Actual Serve	3	0	0	37

Conclusion

Free Form Visualization

The justification section presented a confusion matrix showing the categories of prediction errors that occurred with the final model. I chose to visualize the confusion matrix with an example clip from for each cell in the matrix. A placeholder was substituted for combinations that contained no examples.

Figure 13: Confusion Matrix Visualization

	Predicted Forehand	Predicted Backhand	Predicted Volley	Predicted Serve
Actual Forehand				
Actual Backhand			No Video Available	
Actual Volley		No Video Available		
Actual Serve		No Video Available	No Video Available	

Reflection

This project wasn't easy. At times I thought my model would never learn. When problems arose, it was difficult to determine if the cause was labeling, preprocessing, training, or if there wasn't enough data in the first place. A major breakthrough, and huge morale boost, came when I refactored feature extraction into a preprocessing step. The fact that training time improved by orders of magnitude reinvigorated me to see the project through. Shortly after, I experimented with the additional LSTM layer that made my model learn.

Building a dataset is hard. I vastly underestimated how much work would be required to efficiently create a high quality labeled dataset. Easily half of the work was either labeling itself or building the web page to speed up the process. Bootstrapping the dataset with the model also presented dependency problems. For example, because the model didn't learn initially, I hypothesized that shots from players in the far court were not pronounced enough for the classifier to detect. I therefore relabeled the entire dataset to be

stricter about what constituted a usable forehand, backhand, volley, or serve. This, however, was done without evidence because I didn't have a working model.

Needless to say I didn't anticipate these issues, but I'm glad I faced them. Until my model learned, weeks after I started, I had no idea if this was going to work. I feel proud for choosing a challenging problem and not giving up.

Improvement

The "Model Evaluation and Results" section showed that the final model was overfitting the dataset. To solve this I could either add more dropout to the model or add more training data. Adding more dropout to the model could only theoretically increase the accuracy to that of the training accuracy. Adding training data, while time consuming, seems like the only way to substantially improve the accuracy to above 90% overall accuracy.

Labeling data was time consuming and would probably be the bottleneck in improving the models accuracy. A separate binary classifier could help separate usable vs. unusable clips from the original video reducing the number of clips humans need to view. Spectrograms, visual representations of the audio from a clip, were excluded from the shot classifier because it is very difficult to listen to a clip and distinguish between the type of shot. Spectrograms would be useful, however, to distinguish between clips that have shots and ones that don't. Ultimately if I resume development, a clip filter is the next thing I would work on.

Works Cited

Harvey, Matt. "Continuous Video Classification with TensorFlow, Inception and Recurrent Nets." *Hacker Noon*. Hacker Noon, 30 Dec. 2016. Web. 18 June 2017.