StrokeDetect

A Deep Learning Tennis Stroke Classification Algorithm

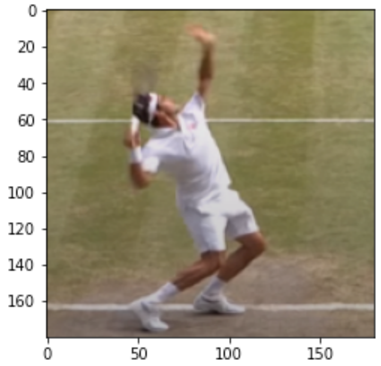
Introductory Question

If I have an image of a tennis player in his/her shot motion, how would I detect what stroke they are hitting? Is it possible to automate this detection process?

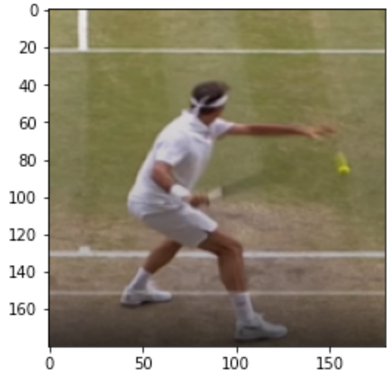
Source

Approach: Image Recognition

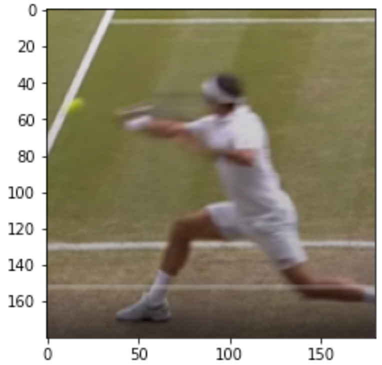
- Since tennis strokes are considered image data, we can recognize them via a convolutional neural network
- To keep the problem simple, let's consider three tennis strokes:
  - Forehand (includes volleys)
  - Backhand (includes volleys)
  - Serve
- We can collect images from a uniform dataset of tennis shots and label them as one of the above (3 classes)
- It would then be possible to implement a deep learning network with TensorFlow 2.3, which can be trained to distinguish between forehand, backhand, and serve.



Serve



Forehand



Backhand

Data Collection

I STARTED BY LOOKING FOR PREMADE TENNIS SHOT DETECTION DATASETS ON KAGGLE, BUT I COULDN'T FIND ANYTHING

I CURATED MY OWN DATASET FROM THE 2019 WIMBLEDON CHAMPIONSHIPS, SPECIFICALLY THE FINAL MATCH BETWEEN FEDERER AND DJOKOVIC

THE DATASET I ASSEMBLED CONTAINS 334 IMAGES WITH ROUGHLY 110 PER CLASS

SMALLER DATASET MAY HAVE LED TO A DEGREE OF OVERFITTING (WHICH WE WILL SEE LATER)

Data Preprocessing (Pt. 1)

- After I collected the 334 images, I separated them into different folders based on shot.
- The next step was to set the batch size (number of samples viewed before model updates) and standardize the image height and width:

Set the batch size, image height and width

```
[ ] batch_size = 32
    img_height = 180
    img_width = 180
    data_dir = '/content/gdrive/My Drive/tennis'
```

Data Preprocessing (Pt. 2)

- The next step is to setup the training and validation datasets. I used a split of 0.8 since it is good practice.
- The batch size was chosen to be 32 since it is good practice for a smaller dataset with less than 100 epochs of training

Setup validation dataset

```
[ ] val_ds = tf.keras.preprocessing.image_dataset_from_directory(
      data_dir,
      validation_split=0.2,
      subset="validation",
      seed=123,
      image_size=(img_height, img_width),
      batch_size=batch_size)
```

```
Found 334 files belonging to 3 classes.
Using 66 files for validation.
```

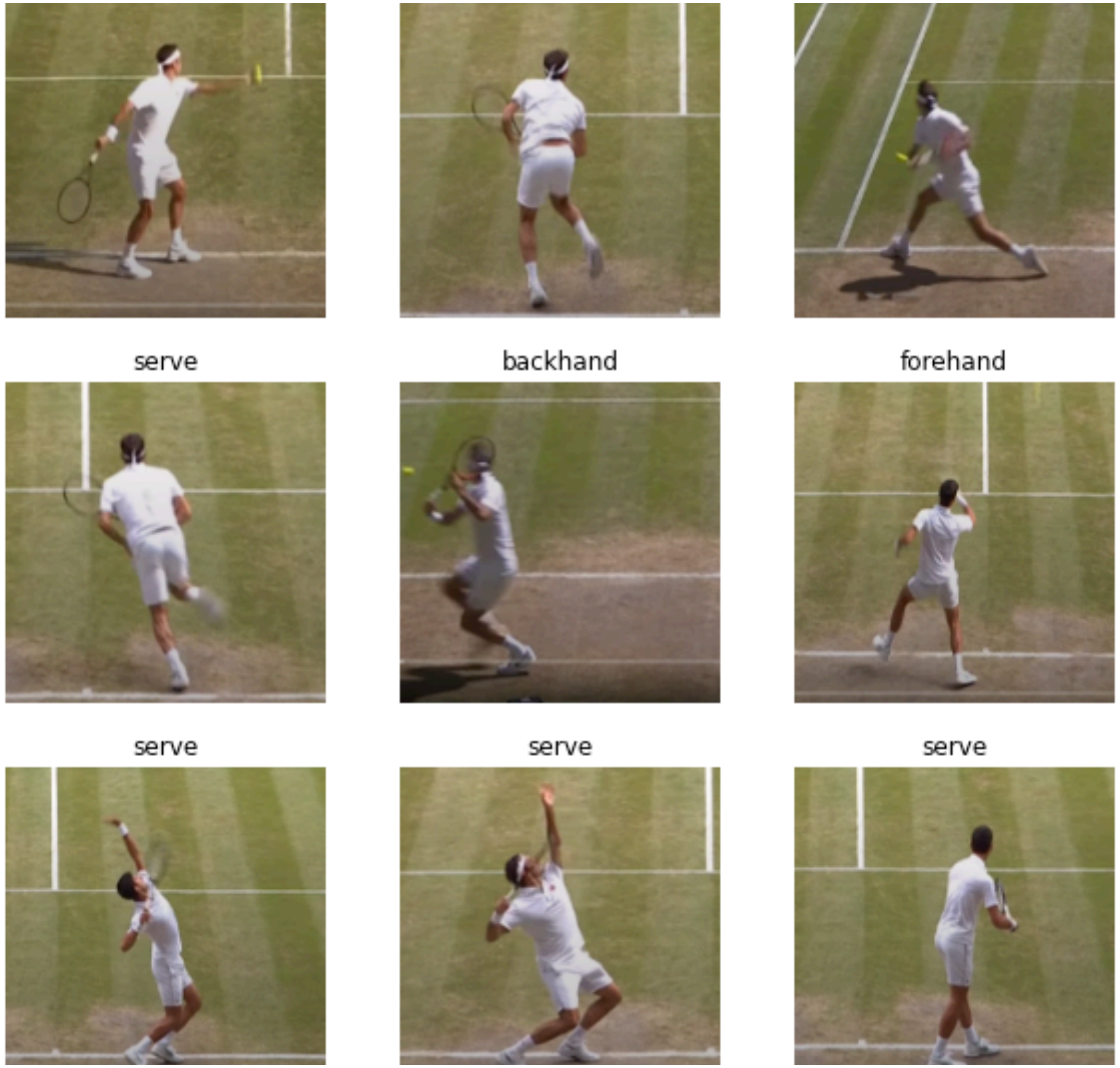
Setup the training dataset

```
[ ] train_ds = tf.keras.preprocessing.image_dataset_from_directory(
      data_dir,
      validation_split=0.2,
      subset="training",
      seed=123,
      image_size=(img_height, img_width),
      batch_size=batch_size)
```

```
Found 334 files belonging to 3 classes.
Using 268 files for training.
```

Data Preprocessing (Pt. 3)

We can now visualize our dataset with matplotlib:



Model Development (Pt. 1)

- After cleaning and visualizing our training data, I developed my network
- I used TensorFlow's Keras API and its sequential model
- We start by defining a normalization layer and setting up our batches of images and labels.
- Pixel values have already been normalized to between 0 and 1

```
[ ] from tensorflow import keras
    from tensorflow.keras import layers
    from tensorflow.keras.models import Sequential

    normalization_layer = layers.experimental.preprocessing.Rescaling(1./255)
```

```
[ ] normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
    image_batch, labels_batch = next(iter(normalized_ds))
    first_image = image_batch[0]
    # Notice the pixels values are now in `[0,1]`.
    print(np.min(first_image), np.max(first_image))
```

```
0.028096406 1.0
```

Model Development (Pt. 2)

- We now define the main sequential model.
- Our model consists of three convolution blocks with a max pool layer in each of them.
- There's a fully connected layer with 128 units on top of it and a `ReLU` activation function.

```
[ ] num_classes = 3

    model = Sequential([
      layers.experimental.preprocessing.Rescaling(1./255),
      layers.Conv2D(16, 3, padding='same', activation='relu'),
      layers.MaxPooling2D(),
      layers.Conv2D(32, 3, padding='same', activation='relu'),
      layers.MaxPooling2D(),
      layers.Conv2D(64, 3, padding='same', activation='relu'),
      layers.MaxPooling2D(),
      layers.Dropout(0.2),
      layers.Flatten(),
      layers.Dense(128, activation='relu'),
      layers.Dense(num_classes)
    ])
```

Model Training (Pt. 1)

- To train the model, we use `tf.GradientTape()` and record loss values with `SparseCategoricalCrossentropy`.
- The optimizer we are using is `Adam` and the learning rate is 0.0003.
- The metric we are using is `SparseCategoricalAccuracy()`

```
[ ] # Instantiate an optimizer to train the model.
    optimizer = keras.optimizers.Adam(learning_rate=0.0003)
    # optimizer = keras.optimizers.SGD(learning_rate=1e-3)

    # Instantiate a loss function.
    loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)

    # Prepare the metrics.
    train_acc_metric = keras.metrics.SparseCategoricalAccuracy()
    val_acc_metric = keras.metrics.SparseCategoricalAccuracy()
```

```
[ ] @tf.function
    def train_step(x, y):
        with tf.GradientTape() as tape:
            logits = model(x, training=True)
            loss_value = loss_fn(y, logits)
        grads = tape.gradient(loss_value, model.trainable_weights)
        optimizer.apply_gradients(zip(grads, model.trainable_weights))
        train_acc_metric.update_state(y, logits)
        return loss_value
```

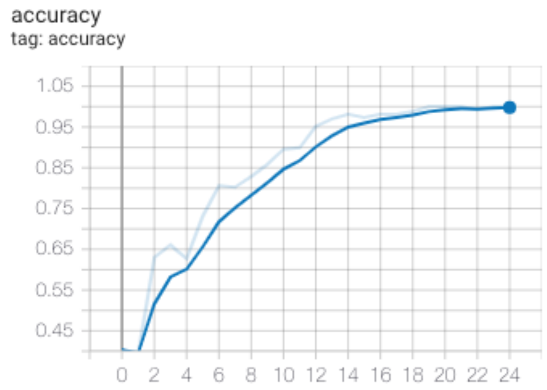
Model Training (Pt. 2)

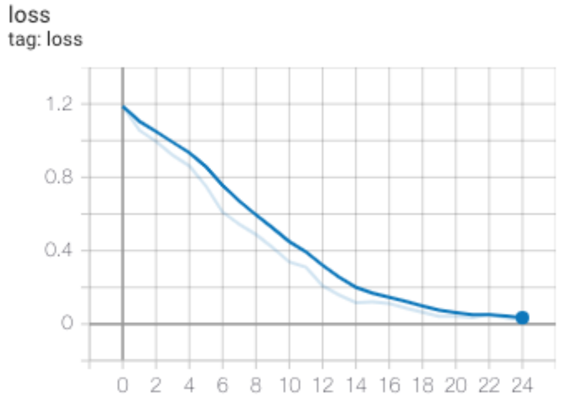
- We can train our model for 24 epochs
- We iterate over batches of the data size and compare our predicted labels to the true labels
  - We add the current batch loss and update during this process
- At the end of every epoch, we display accuracy, loss, and validation accuracy (at the end)

```python
import time

# Keep results for plotting
train_loss_results = []
train_accuracy_results = []

epochs = 20
for epoch in range(epochs):
    print("\nStart of epoch %d" % (epoch,))
    start_time = time.time()

    epoch_loss_avg = tf.keras.metrics.Mean()
    epoch_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()

    # Iterate over the batches of the dataset.
    for step,  (x_batch_train, y_batch_train) in enumerate(train_ds):
        loss_value = train_step(x_batch_train, y_batch_train)

        # Track progress
        epoch_loss_avg.update_state(loss_value)  # Add current batch loss
        # Compare predicted label to actual label
        # training=True is needed only if there are layers with different
        # behavior during training versus inference (e.g. Dropout).
        epoch_accuracy.update_state(y_batch_train, model(x_batch_train, training=True))

    with train_summary_writer.as_default():
      tf.summary.scalar('loss', epoch_loss_avg.result(), step=epoch)
      tf.summary.scalar('accuracy', epoch_accuracy.result(), step=epoch)


    # Display metrics at the end of each epoch.
    train_acc = train_acc_metric.result()
    print("Training acc over epoch: %.4f" % (float(train_acc),))
    print("Epoch {:03d}: Loss: {:.3f}, Accuracy: {:.3%}".format(epoch,
                                                                epoch_loss_avg.result(),
                                                                epoch_accuracy.result()))

    train_loss_results.append(epoch_loss_avg.result())
    train_accuracy_results.append(epoch_accuracy.result())

    # Reset training metrics at the end of each epoch
    train_acc_metric.reset_states()
    epoch_loss_avg.reset_states()
    epoch_accuracy.reset_states()

    # Run a validation loop at the end of each epoch.
    for x_batch_val, y_batch_val in val_ds:
        test_step(x_batch_val, y_batch_val)

    val_acc = val_acc_metric.result()
    val_acc_metric.reset_states()
    print("Validation acc: %.4f" % (float(val_acc),))
    print("Time taken: %.2fs" % (time.time() - start_time))
```

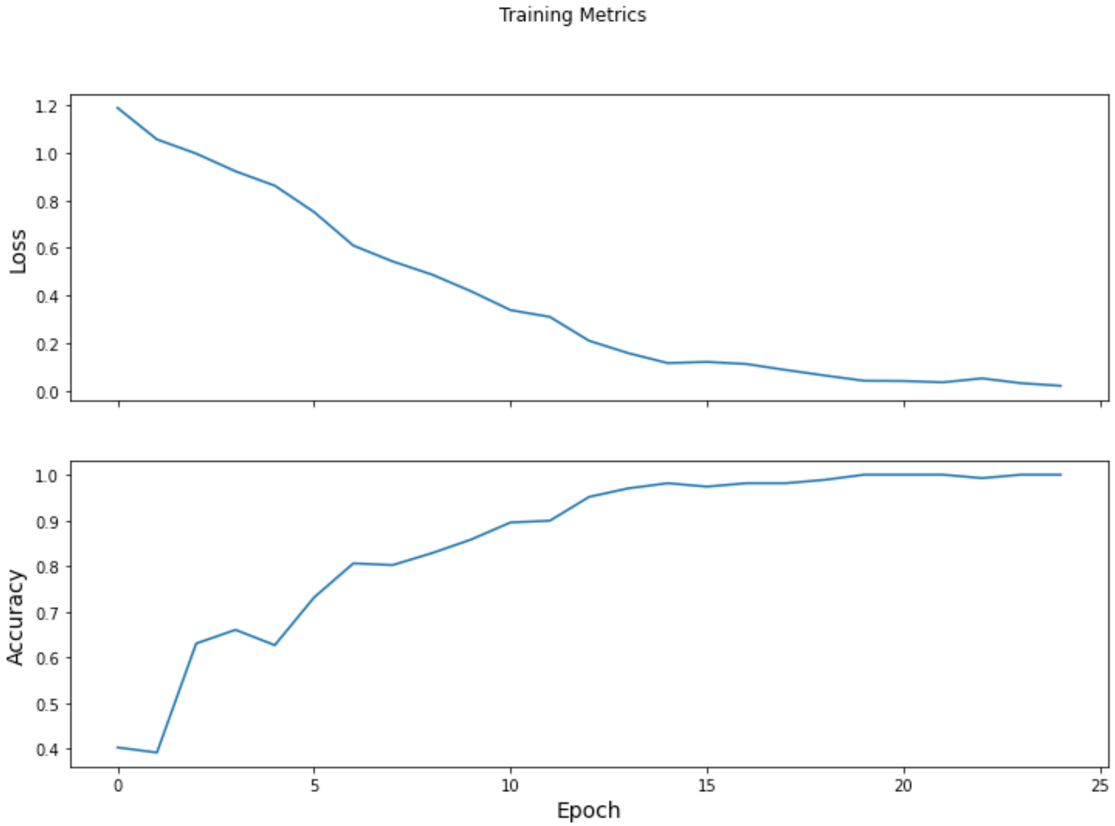
Results (Pt. 1)

- As you can see in the graphs below, our model behavior was correct in that loss decreases accuracy increases over the epochs
- By the final epoch (24), the loss was 0.021 and the accuracy was 100.00%





Results (Pt. 2)

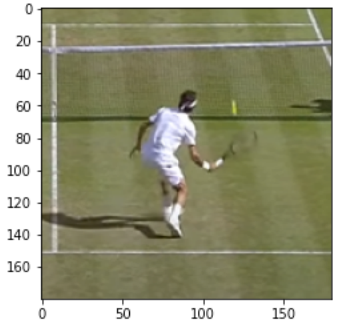


Training Metrics
Loss
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Accuracy
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0
5
10
15
20
25
Epoch

Inference Results

```python
test_path = "/content/gdrive/My Drive/test/test2.png"


img1 = PIL.Image.open(test_path)
img1_resized = img1.resize((180, 180))
plt.imshow(img1_resized)

img = keras.preprocessing.image.load_img(
    test_path, target_size=(img_height, img_width)
)
img_array = keras.preprocessing.image.img_to_array(img)
img_array = tf.expand_dims(img_array, 0) # Create a batch

predictions = model.predict(img_array)
score = tf.nn.softmax(predictions[0])

print(
    "This image most likely belongs to {} with a {:.2f} percent confidence."
    .format(class_names[np.argmax(score)], 100 * np.max(score))
)
```

```
This image most likely belongs to forehand with a 99.04 percent confidence.
```



Conclusions and Future Work

- In the future, I hope to add to my project by expanding the dataset.
- When you try inference for shots not from Wimbledon, the performance drops, so I want to add in 250 images at minimum for all the major surfaces (grass, hard, clay)
- I also want to add in more players. Right now, the model is thrown off by left-handed players, so incorporating lefties like Rafael Nadal will be interesting and will require a large dataset.
- To increase the sophistication of the algorithm, I would like to try other existing pre-trained CNNs as well (ResNet, VGG, etc.)