HES-SO — Machine Learning — PW 14

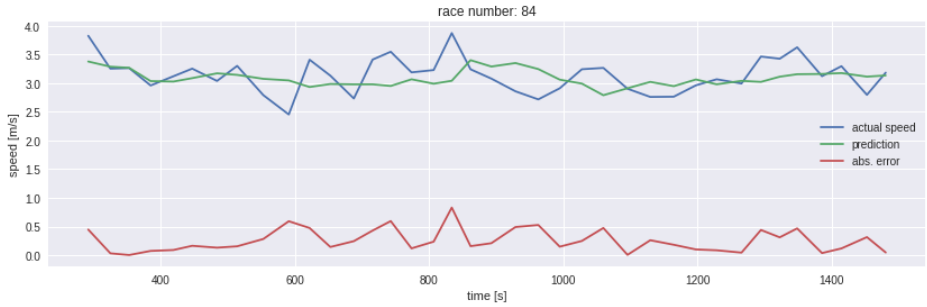
January 7, 2019

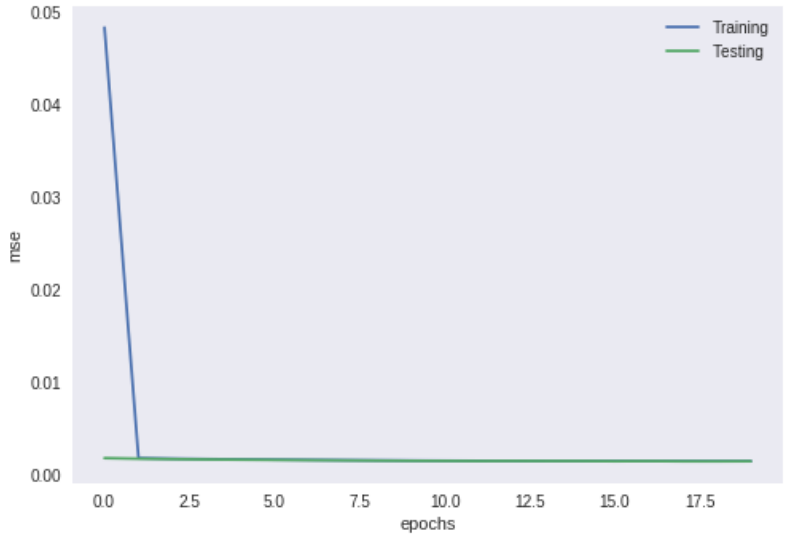
This document has been written by the following members of the lecture: Claret Romain, Ménétrey Jämes, Rochat Damien and Sandoz Michaël.

0.1 Change the number of units and epochs of the LSTM network. Show the configuration that performed the best.

The initial result of the training with the batch size of **64**, number of epochs set to **20** and the number of LSTM units set to **1**:

- Training correlation coefficient: 0.639540111378689
- Test correlation coefficient: 0.6416686449673735





We noticed that the initial results are showing a sign of overfitting, indeed the correlation coefficient on the testing set performs better than on the training set. The batch size is probably too small.

The following code has been used to find an optimal number of LSTM units using a systematic approach. While experimenting we noticed a few points:

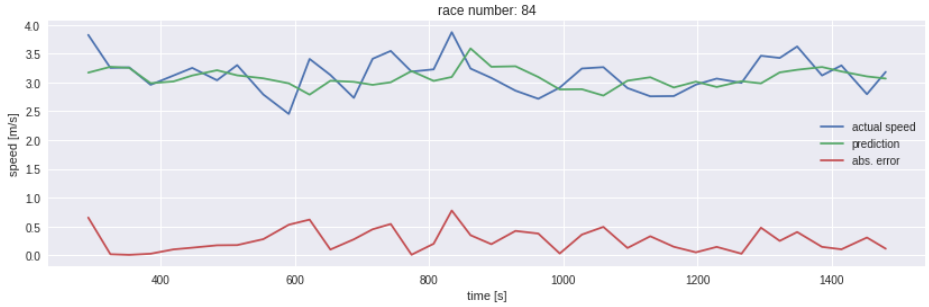
- We used mostly the TPU backend on Google Colab because it's performing faster, about a ratio of **1:4**. Concerning result comparison from TPU and GPU, we noticed that GPU equivalent results are generally underfitted or overfitted.
- Running the initial code on a TPU backend is providing a different results, which is not overfitting. Indeed, the value for the training correlation coefficient is **0.6252519179188435** and for the testing set **0.6235112426366156**.
- We are running each experiment 3 times and keep the best testing result.

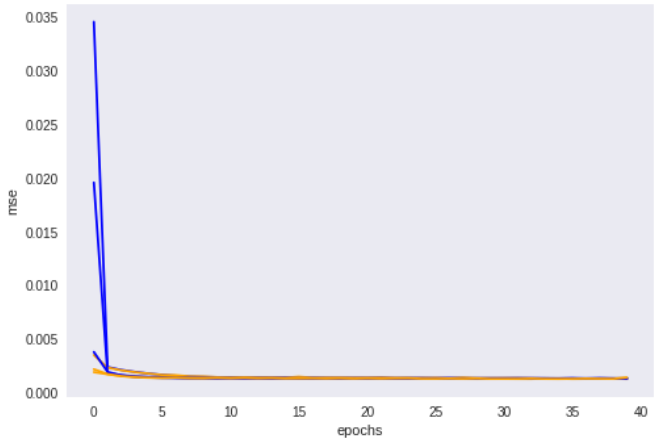
```
In [ ]: BATCH_SIZE = 140
        NB_EPOCHS = 40
        NB_UNITS = 20

        h_train = []
        h_test = []

        REPEAT = 3
        for i in range(1, 2, 1):
          start = timer()
          for j in range (1,REPEAT+1,1):
            print("progress ", j, "/", REPEAT)

            #BATCH_SIZE = i
            #NB_EPOCHS = i
            #NB_UNITS = i

            model = Sequential()
            model.add(LSTM(NB_UNITS, input_shape=(TIMESTEPS, len(FEATURES))))
            model.add(Dense(1))

            model.compile(loss='mean_squared_error', optimizer='adam')

            history = model.fit(X_train, y_train, epochs=NB_EPOCHS,
             ↪  batch_size=BATCH_SIZE, verbose=0, validation_data=(X_test, y_test))

            y_train_pred = model.predict(X_train)
            y_test_pred = model.predict(X_test)

            h_train.append(np.corrcoef(y_train.T, y_train_pred.T)[0,1])
            h_test.append(np.corrcoef(y_test.T, y_test_pred.T)[0,1])

            # Plot the training and testing
            pl.plot(history.history['loss'], color='blue')
            pl.plot(history.history['val_loss'], color='orange')

            #print(history.history['loss'][:-1])

          print("BATCH_SIZE = ", BATCH_SIZE)
```

```
print("NB_EPOCHS = ", NB_EPOCHS)
print("NB_UNITS = ", NB_UNITS)
print("Best correlation coefficient for training and testing
↪  sets:",h_train[np.argmax(h_test)],h_test[np.argmax(h_test)])


pl.xlabel('epochs')
pl.ylabel('mse')
#pl.legend()
pl.grid()
pl.show()

end = timer()
print(end - start, "seconds") # Time in seconds

h_train = []
h_test = []

print()
print("-----------")
print()
```

We ran multiple experiments and we were filling an excel spreadsheet to help us optimize the results. We chose the parameters to be the most performant and the less time consuming. Indeed, we found that using a batch size of **140** and **20** LSTM units was performing well. Concerning the epoch amount, more we are presenting the data, better it performs, however, we observed that **40** performs well for a low learning time, about 1.5 minutes.

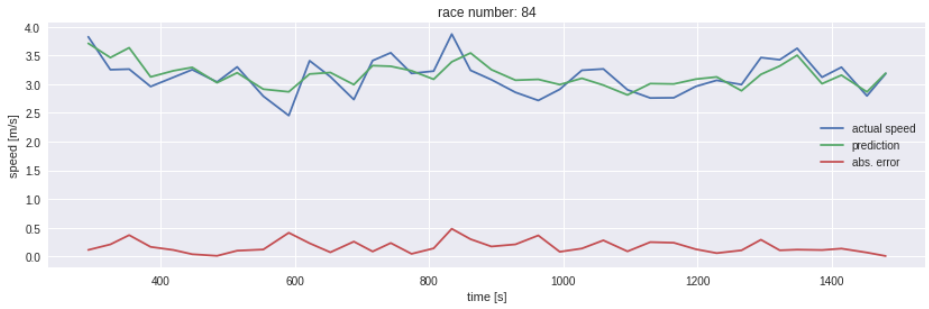
- TPU Training correlation coefficient: 0.6553012087885829
- TPU Test correlation coefficient: 0.6478590610500015

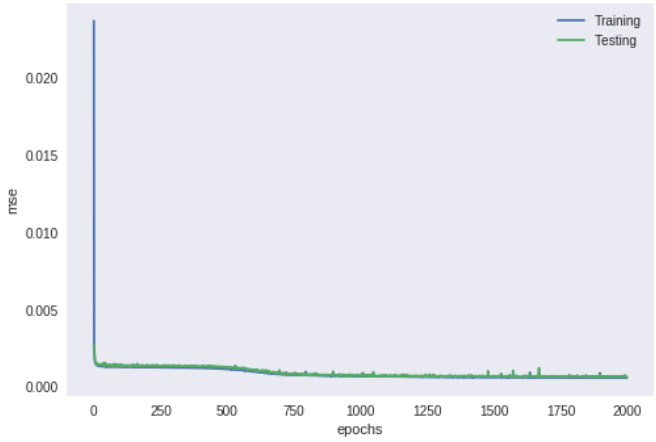




About 1 hour later, using **2000** epochs:

- TPU Training correlation coefficient: 0.8651456135127776
- TPU Test correlation coefficient: 0.8555605818219801





0.2 What is the largest error (speed prediction) you observed? Do you observe that most of those large errors show up for high speeds ? or low speeds? Why?

The following code highlights the largest error (MSE):

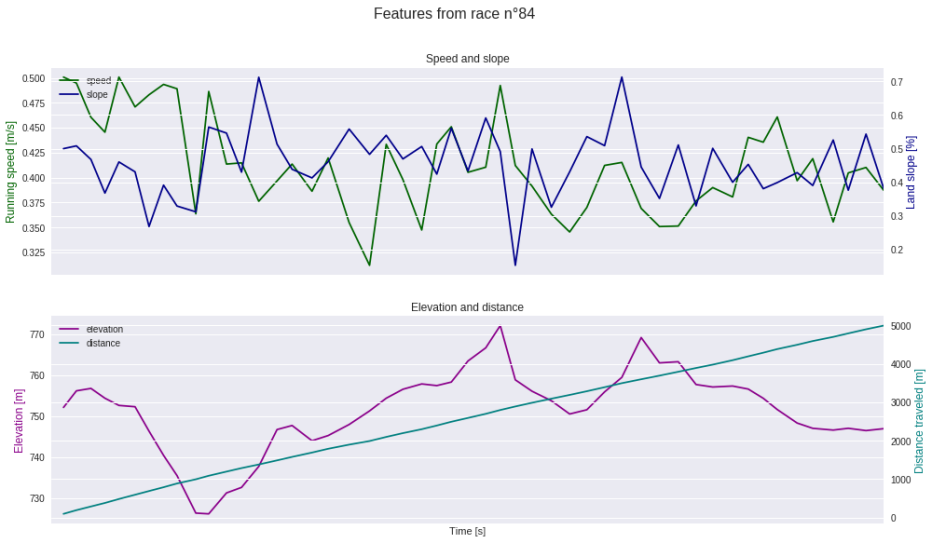
```
In [ ]: mse = np.abs(y_o - y_pred_o[:,0])
        max_error_index = np.argmax(mse)
        max_error_value = np.max(mse)
```

```
print("max_error_index:", max_error_index)
print("max_error_value:", max_error_value)
print("actual speed (at the max):", y_o[max_error_index])
print("predicated speed (at the max):", y_pred_o[max_error_index])
```

The output of the code using the optimized parameters are the following:

- max_error_index: 17
- max_error_value: 0.4829763010962034
- actual speed (at the max): 3.871952636621716
- premeditated speed (at the max): 3.3889763

The most large errors occur when the actual speed have high picks. The intuitive explanation is because the course first has low speed, thus the LSTM units memorize this result. The second part of the course has an increase of speed and the LSTM units are not yet trained to handle such scenario. If we map the output with the course (presented below), we can see the speed is inversely proportional to the elevation.



0.3 Compute the correlation between the next speed (model output) and the current speed (model input). Does your LSTM perform better than just using the current speed as a prediction of the next speed ?

The correlation is computed using the mean squared error (lower is better). In order to compare the current speed and the next speed, we need to compare the element y_i with y_{i+1}. The following code illustrates the comparison:

```
In [ ]: def mse(A, B):
            return (np.square(A - B)).mean()

        print(mse(y_o[:-1], y_o[1:]))
```

The output is: 0.13286001427182506. The performance of the LSTM is computed as follows:

```
In [ ]: print(mse(y_o, y_pred_o))
```

The output is 0.13739563085444073. Therefore, using the current speed as a prediction of the next speed is better than using the LSTM in this instance.

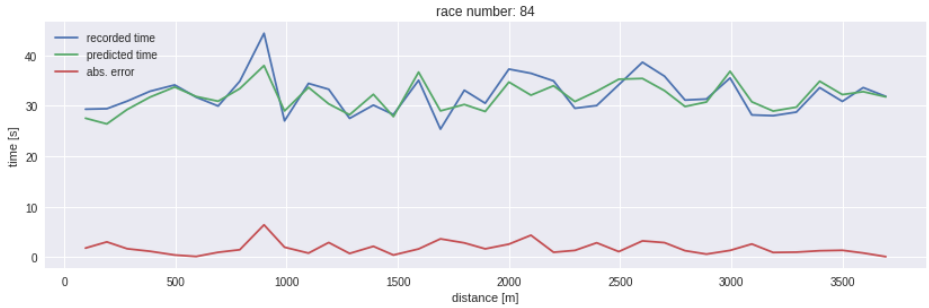
0.4 Using the predicted speeds for a given race, compute the expected time for a race and compute the difference between the real race time and the predicted race time in minutes. Provide the code of the cell that computes this prediction error.

```
In [ ]: def get_distances(data, races):
          for r in races:
            race_df = data.loc[data['race'] == r]
            race_np = race_df['distance'].values
            race_np = [race_np[i:(i+TIMESTEPS+1)] for i in range(race_np.shape[0] -
             ↪  (TIMESTEPS+2))]
            if len(race_np) == 0:
              print("Warning: not enough values in race", r)
              continue

            race_np = np.stack(race_np, axis=0)

          return race_np


        X, y = create_x_y(dataset, [random_race])
        X_o, y_o = create_x_y(original_dataset, [random_race])

        y_pred_o = model.predict(X) * (max_speed - min_speed) + min_speed

        x_o_dist = get_distances(dataset, [random_race])

        time_recorded = 0
        time_predicted = 0
        times_recorded = []
        times_predicted = []
        dists_recorded = [0]

        for i in range(1,len(y_o)):
          tmp_recorded_dist = x_o_dist[:,0][i]-x_o_dist[:,0][i-1]
          dists_recorded.append(tmp_recorded_dist+dists_recorded[-1])

          tmp_recorded_speed = y_o[i]
          tmp_recorded_time = tmp_recorded_dist/tmp_recorded_speed
          time_recorded += tmp_recorded_time
          times_recorded.append(tmp_recorded_time)

          tmp_predicted_speed = y_pred_o[i][0]
          tmp_predicted_time = tmp_recorded_dist/tmp_predicted_speed
          time_predicted += tmp_predicted_time
          times_predicted.append(tmp_predicted_time)

        pl.figure(figsize=(14,4))
        pl.plot(dists_recorded[1:], times_recorded, label='recorded time')
        pl.plot(dists_recorded[1:], times_predicted, label='predicted time')
```

```
pl.plot(dists_recorded[1:], np.abs(np.array(times_recorded) -
 ↪  np.array(times_predicted)), label='abs. error')
pl.legend()
pl.title('race number: ' + str(random_race))
pl.xlabel('distance [m]')
pl.ylabel('time [s]');

print("recorded time:", time_recorded/60, "minutes")
print("predicted time:", time_predicted/60, "minutes")
print("abs. time error:", np.abs(time_recorded - time_predicted)/60, "minutes")
```

We get the following result:

- recorded time: 19.880892867664276 minutes
- predicted time: 19.60883281109609 minutes
- abs. time error: 0.2720600565681858 minutes



To be able to compute the time, we had to compute the distance. Indeed, we computed the distance interval with the last record/predicted speed.