[Jupyter Notebook](http://localhost:8889/tree?token=cda4af3e3e5c74995a83b6cf84b01cb569f4745727e4dc9d)

Deep\_LearningLast Checkpoint: Last Saturday at 5:42 PMAutosave Failed!Logout

tf\_keras\_gpu

Trusted

* [File](http://localhost:8889/notebooks/Documents/GitHub/Springboard/Capstone_Project_2/notebooks/Deep_Learning.ipynb)
* [Edit](http://localhost:8889/notebooks/Documents/GitHub/Springboard/Capstone_Project_2/notebooks/Deep_Learning.ipynb)
* [View](http://localhost:8889/notebooks/Documents/GitHub/Springboard/Capstone_Project_2/notebooks/Deep_Learning.ipynb)
* [Insert](http://localhost:8889/notebooks/Documents/GitHub/Springboard/Capstone_Project_2/notebooks/Deep_Learning.ipynb)
* [Cell](http://localhost:8889/notebooks/Documents/GitHub/Springboard/Capstone_Project_2/notebooks/Deep_Learning.ipynb)
* [Kernel](http://localhost:8889/notebooks/Documents/GitHub/Springboard/Capstone_Project_2/notebooks/Deep_Learning.ipynb)
* [Widgets](http://localhost:8889/notebooks/Documents/GitHub/Springboard/Capstone_Project_2/notebooks/Deep_Learning.ipynb)
* [Help](http://localhost:8889/notebooks/Documents/GitHub/Springboard/Capstone_Project_2/notebooks/Deep_Learning.ipynb)

Run



**SpringBoard Capstone Project 2: Human Activity Recognition**

**Deep Learning**

**Written by: Sofy Weisenberg**  
**Date: 04/21/20**

**Importing the Data**  
Load the data from file into a pandas dataframe. In previous sections, the dataset used had undergone feature engineering (with 9 signal time series transformed into 561 features for each 2.56 s window of the time series). Implementing a deep learning approach will require switching to the raw data in order to allow the network to find its own features in the data.

Therefore, the raw data will need to be loaded into an appropriately sized feature array X. The label vector y from previous sections remains the same.

Each row/observation still represents a single 2.56 s window, and these windows still have 50% overlap per the design of the collected data. However, now each row contains 128 time steps of raw signal data (50 Hz x 2.56 s). A row with 128 columns exists for each of the 9 signal channels:

* body acceleration (x,y,z)
* body gyroscope (x,y,z)
* total acceletation (x,y,z)

Therefore, when combining all the signal data, each row should actually have 128 x 9 = 1152 columns or features. The 9 channels of data are stored in 9 separate files and are already separated into training and test sets (see "Splitting the Data: Training and Test Sets" in the previous section).

**Categorical Class Encoding**

Since the class labels (Activity column) for this data are in a categorical string format, they must first be encoded to a numerical format useable for supervised machine learning. Simple integer encoding (i.e. translating each class label into an integer label 1-6) would not be effective on its own in this case, since such an encoding implies an ordinal relationship between the classes, where in reality there is none. This may result in poor model performance or unexpected results.

Integer encoding followed by one-hot encoding is the preferred method to transform the categorical data. Each integer class label is assigned to a new binary (0/1) column of "dummy variables". Each observation in the dataset is then be labeled with a 1 in only one of these "dummy variable" columns and the rest are labeled with a 0, resulting in sparse matrices of class labels for y\_train and y\_test.

In [2]:



**import** numpy **as** np

**import** pandas **as** pd

**from** sklearn.decomposition **import** PCA

**from** sklearn.manifold **import** TSNE

**import** matplotlib.pyplot **as** plt

**from** mpl\_toolkits.mplot3d **import** Axes3D

**import** seaborn **as** sns

**import** os

**import** time

**import** tensorflow **as** tf

**from** tensorflow **import** keras

​

*# Use the current working directory path to navigate to the location of the processed data*

cwd\_path **=** os.getcwd()

data\_path **=** os.path.join(cwd\_path, '..', 'data', 'raw')

​

*# Throw an assert error if the path does not exist*

**assert** os.path.exists(data\_path)

The code for loading the data was modified from the guide found here: <https://machinelearningmastery.com/how-to-develop-rnn-models-for-human-activity-recognition-time-series-classification/>

In [3]:



*# load a single file as a numpy array*

**def** load\_file(filepath):

df **=** pd.read\_csv(filepath, header**=None**, delim\_whitespace**=True**)

**return** df.values

In [4]:



*# load all channels into a stacked 3D array*

**def** load\_files\_3D\_array(filenames, prefix**=**''):

loaded **=** list()

**for** name **in** filenames:

data **=** load\_file(os.path.join(prefix, name))

loaded.append(data)

*# stack array so that features are the 3rd dimension*

loaded **=** np.dstack(loaded)

**return** loaded

In [5]:



*# load a dataset group, specify group = 'train' or 'test', prefix is location of group data folder*

**def** load\_dataset\_group(group, prefix**=**''):

filepath **=** os.path.join(prefix, group, 'Inertial Signals')

*# list all 9 filenames in group (train/test)*

filenames **=** list()

*# body acceleration*

filenames **+=** ['body\_acc\_x\_'**+**group**+**'.txt', 'body\_acc\_y\_'**+**group**+**'.txt', 'body\_acc\_z\_'**+**group**+**'.txt']

*# body gyroscope*

filenames **+=** ['body\_gyro\_x\_'**+**group**+**'.txt', 'body\_gyro\_y\_'**+**group**+**'.txt', 'body\_gyro\_z\_'**+**group**+**'.txt']

*# total acceleration*

filenames **+=** ['total\_acc\_x\_'**+**group**+**'.txt', 'total\_acc\_y\_'**+**group**+**'.txt', 'total\_acc\_z\_'**+**group**+**'.txt']

*# load 3D stacked feature array for group (train/test)*

X **=** load\_files\_3D\_array(filenames, filepath)

*# load class output array for group (train/test)*

y **=** load\_file(os.path.join(prefix, group, 'y\_'**+**group**+**'.txt'))

**return** X, y

In [6]:



**from** tensorflow.keras.utils **import** to\_categorical

​

*# load the entire dataset, prefix is filepath of dataset folder*

*# returns train and test stacked feature array X and class output array y (one-hot encoded) for train, test groups*

**def** load\_dataset(prefix**=**''):

*# load all training set*

X\_train, y\_train **=** load\_dataset\_group('train', prefix)

*# load all test set*

X\_test, y\_test **=** load\_dataset\_group('test', prefix)

*# zero-offset class values (transform integers 1-6 to integers 0-5 for one-hot encoding)*

y\_train **=** y\_train **-** 1

y\_test **=** y\_test **-** 1

*# one hot encode y*

y\_train **=** to\_categorical(y\_train)

y\_test **=** to\_categorical(y\_test)

print('Training feature array X size, training label vector y size:')

print(X\_train.shape, y\_train.shape)

print('\nTest feature array X size, test label vector y size:')

print(X\_test.shape, y\_test.shape)

**return** X\_train, y\_train, X\_test, y\_test

In [7]:



X\_train, y\_train, X\_test, y\_test **=** load\_dataset(data\_path)

Training feature array X size, training label vector y size:

(7352, 128, 9) (7352, 6)

Test feature array X size, test label vector y size:

(2947, 128, 9) (2947, 6)

**Feature Scaling**  
To improve performance of the machine learning model(s) to be applied to this classification problem, first the feature data should be appropriately scaled.

Scaling must be performed both on the training and test data sets in the same manner. Best practice is to use only the training set to identify the correct scaling, and then blindly apply the same transformation to the test set. Scaling all features to a [0,1] range is a common scaling method for neural network preprocessing and may be applied using the MinMaxScaler() transformation method from sklearn.

However, after evaluating the model with [0,1] scaling, poor performance was observed (approx. 0.78 accuracy). MinMaxScaler() with range [1,1] scaling performed much better (approx. 0.89 accuracy) and StandardScaler() with mean centered around 0 and std. dev = 1 performed even slightly better (approx. 0.91 accuracy). Therefore, the StandardScaler() method was selected for feature scaling.

In [8]:



**from** sklearn.preprocessing **import** MinMaxScaler

**from** sklearn.preprocessing **import** StandardScaler

*# initialize scaled feature arrays*

X\_train\_scaled **=** np.empty(X\_train.shape)

X\_test\_scaled **=** np.empty(X\_test.shape)

​

*# initialize scaler function*

*#scaler = MinMaxScaler(feature\_range=(-1, 1))*

scaler **=** StandardScaler()

​

*# scaler only accepts 2D arrays, so loop over 9 2D arrays to fit and transform accordingly for each*

**for** i **in** range(X\_train.shape[2]):

scaler.fit(X\_train[:,:,i])

X\_train\_scaled[:,:,i] **=** scaler.transform(X\_train[:,:,i])

X\_test\_scaled[:,:,i] **=** scaler.transform(X\_test[:,:,i])

The scaling transformation has been performed successfully on both the training and test feature sets.

* The mean for each column in the X\_train\_scaled feature array is 0, and std. dev is 1.
* The mean for each column in the X\_test\_scaled feature array is NOT 0, and std. dev is NOT 1, since a transformation matrix (generated from normalizing the training feature matrix range) was applied to the test feature matrix.

**Fitting an RNN Model**  
Now that the data has been loaded and preprocessing completed, a deep learning model may be trained on the data. The first model that will be implemented is a Recurrent Neural Network (RNN) which leverages the sequential nature of the time series signal data from the 9 different channels. Specifically, the LSTM (long short term memory) layer of the neural network, unlike feed forward neural network architectures, has feedback mechanisms that allow previous observations in the sequence to be "remembered" by the network.

First, a basic network architecture will be established. Then several network hyperparameters will be optimized using a grid search optimization.

In [11]:



*# Use scikit-learn to grid search the network hyperparameters*

**from** sklearn.model\_selection **import** GridSearchCV

**from** tensorflow.keras.models **import** Sequential

**from** tensorflow.keras.layers **import** Dense

**from** tensorflow.keras.layers **import** Dropout

**from** tensorflow.keras.layers **import** LSTM

**from** tensorflow.keras.wrappers.scikit\_learn **import** KerasClassifier

**from** tensorflow.keras.callbacks **import** EarlyStopping

​

*# Function to create model, required for KerasClassifier*

​

**def** create\_RNN\_model(neurons**=**100, dropout**=**0.5, batch\_size**=**64):

*# Define key array lengths*

n\_timesteps, n\_features, n\_outputs **=** X\_train.shape[1], X\_train.shape[2], y\_train.shape[1]

*# Define input layer and hidden layers*

model **=** Sequential()

model.add(LSTM(neurons, input\_shape**=**(n\_timesteps, n\_features)))

model.add(Dropout(dropout))

model.add(Dense(neurons, kernel\_initializer**=**'glorot\_uniform', activation**=**'relu'))

*# Define output layer with multiple outputs and compile model*

model.add(Dense(n\_outputs, activation**=**'softmax'))

model.compile(loss**=**'categorical\_crossentropy', optimizer**=**'adam', metrics**=**['accuracy'])

**return** model

In [12]:



*# initialize model and early stopping monitor (if model convergence reached before n\_epochs)*

n\_epochs **=** 35

model **=** KerasClassifier(build\_fn**=**create\_RNN\_model, verbose**=**0, epochs**=**n\_epochs)

es **=** EarlyStopping(monitor**=**'loss', mode**=**'min', verbose**=**1, patience**=**5)

​

*# define the grid search parameters*

neurons **=** [50, 100, 200]

dropout **=** [0.1,0.3,0.5]

batch\_size **=** [32, 64, 128]

​

param\_grid **=** dict(neurons**=**neurons,

dropout**=**dropout,

batch\_size**=**batch\_size)

​

*# setup grid search*

grid **=** GridSearchCV(estimator**=**model, param\_grid**=**param\_grid, cv**=**3, verbose**=**1, n\_jobs**=**1)

In [73]:



*# fit the grid search models and report results with early stopping callback*

grid\_result **=** grid.fit(X\_train\_scaled, y\_train, callbacks**=**[es])

Fitting 3 folds for each of 27 candidates, totalling 81 fits

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

Epoch 00015: early stopping

Epoch 00017: early stopping

Epoch 00016: early stopping

Epoch 00021: early stopping

Epoch 00018: early stopping

Epoch 00019: early stopping

Epoch 00014: early stopping

Epoch 00018: early stopping

Epoch 00028: early stopping

Epoch 00023: early stopping

Epoch 00016: early stopping

Epoch 00015: early stopping

Epoch 00019: early stopping

Epoch 00028: early stopping

Epoch 00015: early stopping

Epoch 00010: early stopping

Epoch 00028: early stopping

Epoch 00027: early stopping

Epoch 00017: early stopping

Epoch 00032: early stopping

Epoch 00009: early stopping

Epoch 00012: early stopping

Epoch 00014: early stopping

Epoch 00013: early stopping

Epoch 00015: early stopping

Epoch 00017: early stopping

Epoch 00013: early stopping

Epoch 00020: early stopping

Epoch 00014: early stopping

Epoch 00022: early stopping

Epoch 00013: early stopping

Epoch 00016: early stopping

Epoch 00026: early stopping

Epoch 00013: early stopping

Epoch 00016: early stopping

Epoch 00021: early stopping

Epoch 00017: early stopping

Epoch 00022: early stopping

Epoch 00012: early stopping

Epoch 00032: early stopping

Epoch 00022: early stopping

Epoch 00015: early stopping

Epoch 00017: early stopping

Epoch 00030: early stopping

Epoch 00019: early stopping

Epoch 00018: early stopping

Epoch 00020: early stopping

Epoch 00018: early stopping

Epoch 00015: early stopping

Epoch 00028: early stopping

Epoch 00022: early stopping

Epoch 00023: early stopping

Epoch 00022: early stopping

Epoch 00019: early stopping

Epoch 00027: early stopping

Epoch 00018: early stopping

Epoch 00019: early stopping

Epoch 00025: early stopping

Epoch 00023: early stopping

Epoch 00014: early stopping

Epoch 00028: early stopping

Epoch 00031: early stopping

Epoch 00028: early stopping

Epoch 00020: early stopping

Epoch 00015: early stopping

Epoch 00025: early stopping

Epoch 00014: early stopping

Epoch 00016: early stopping

Epoch 00015: early stopping

Epoch 00030: early stopping

Epoch 00016: early stopping

Epoch 00011: early stopping

Epoch 00016: early stopping

Epoch 00025: early stopping

Epoch 00018: early stopping

Epoch 00024: early stopping

Epoch 00013: early stopping

[Parallel(n\_jobs=1)]: Done 81 out of 81 | elapsed: 238.7min finished

Epoch 00020: early stopping

In [42]:



*# summarize grid search results and report best parameter set*

print("Best: %f using %s" **%** (grid\_result.best\_score\_, grid\_result.best\_params\_))

means **=** grid\_result.cv\_results\_['mean\_test\_score']

stds **=** grid\_result.cv\_results\_['std\_test\_score']

params **=** grid\_result.cv\_results\_['params']

**for** mean, stdev, param **in** zip(means, stds, params):

print("%f (%f) with: %r" **%** (mean, stdev, param))

Best: 0.904514 using {'batch\_size': 64, 'dropout': 0.5, 'neurons': 100}

0.904514 (0.030451) with: {'batch\_size': 64, 'dropout': 0.5, 'neurons': 100}

Now that the best parameter set has been identified, a model with these parameters will be fit to the training data and evaluated. The training history will be plotted to visualize the validation accuracy and loss vs. the progression of the epochs. This shows model convergence and can hint at overfitting. A confusion matrix then shows the model's overall performance across classes.

In [13]:



*# Define function to plot the training history (accuracy and loss for the training and validation sets)*

**def** show\_train\_history(train\_history,n\_epochs,train,validation):

plt.plot(train\_history.history[train])

plt.plot(train\_history.history[validation])

plt.title('Train History: %i epochs' **%** n\_epochs)

plt.ylabel(train)

plt.xlabel('Epoch')

plt.xticks(np.arange(0, n\_epochs, step**=**round(n\_epochs**/**10)))

plt.legend(['train', 'validation'], loc**=**'best')

plt.show()

In [76]:



*# initialize a model with the best parameters from the grid search*

start\_time **=** time.time()

model **=** create\_RNN\_model(neurons**=**grid\_result.best\_params\_['neurons'],

dropout**=**grid\_result.best\_params\_['dropout'],

batch\_size**=**grid\_result.best\_params\_['batch\_size'])

​

*# fit the model and output the training history so it may be plotted*

train\_history **=** model.fit(x**=**X\_train\_scaled, y**=**y\_train,

validation\_split**=**0.25,

epochs**=**n\_epochs,

batch\_size**=**grid\_result.best\_params\_['batch\_size'],

verbose**=**2,

callbacks**=**[es])

​

print("--- Training executed in %s seconds ---" **%** (time.time() **-** start\_time))

Train on 5514 samples, validate on 1838 samples

Epoch 1/35

5514/5514 - 14s - loss: 0.9982 - accuracy: 0.6099 - val\_loss: 0.5149 - val\_accuracy: 0.8074

Epoch 2/35

5514/5514 - 12s - loss: 0.4362 - accuracy: 0.8330 - val\_loss: 0.4335 - val\_accuracy: 0.8645

Epoch 3/35

5514/5514 - 12s - loss: 0.3342 - accuracy: 0.8814 - val\_loss: 0.2817 - val\_accuracy: 0.9032

Epoch 4/35

5514/5514 - 12s - loss: 0.2166 - accuracy: 0.9258 - val\_loss: 0.1907 - val\_accuracy: 0.9325

Epoch 5/35

5514/5514 - 12s - loss: 0.1423 - accuracy: 0.9496 - val\_loss: 0.1851 - val\_accuracy: 0.9380

Epoch 6/35

5514/5514 - 12s - loss: 0.1380 - accuracy: 0.9460 - val\_loss: 0.1887 - val\_accuracy: 0.9380

Epoch 7/35

5514/5514 - 12s - loss: 0.1398 - accuracy: 0.9454 - val\_loss: 0.1929 - val\_accuracy: 0.9282

Epoch 8/35

5514/5514 - 12s - loss: 0.1315 - accuracy: 0.9487 - val\_loss: 0.2031 - val\_accuracy: 0.9331

Epoch 9/35

5514/5514 - 12s - loss: 0.1284 - accuracy: 0.9432 - val\_loss: 0.1931 - val\_accuracy: 0.9391

Epoch 10/35

5514/5514 - 12s - loss: 0.1089 - accuracy: 0.9519 - val\_loss: 0.2228 - val\_accuracy: 0.9227

Epoch 11/35

5514/5514 - 12s - loss: 0.0996 - accuracy: 0.9576 - val\_loss: 0.2400 - val\_accuracy: 0.9244

Epoch 12/35

5514/5514 - 13s - loss: 0.1152 - accuracy: 0.9519 - val\_loss: 0.2084 - val\_accuracy: 0.9266

Epoch 13/35

5514/5514 - 13s - loss: 0.1062 - accuracy: 0.9523 - val\_loss: 0.2745 - val\_accuracy: 0.9178

Epoch 14/35

5514/5514 - 12s - loss: 0.0970 - accuracy: 0.9543 - val\_loss: 0.2439 - val\_accuracy: 0.9021

Epoch 15/35

5514/5514 - 12s - loss: 0.1253 - accuracy: 0.9518 - val\_loss: 0.2197 - val\_accuracy: 0.9222

Epoch 16/35

5514/5514 - 12s - loss: 0.0934 - accuracy: 0.9572 - val\_loss: 0.2246 - val\_accuracy: 0.9353

Epoch 17/35

5514/5514 - 12s - loss: 0.1000 - accuracy: 0.9538 - val\_loss: 0.2762 - val\_accuracy: 0.8052

Epoch 18/35

5514/5514 - 12s - loss: 0.1555 - accuracy: 0.9349 - val\_loss: 0.1705 - val\_accuracy: 0.9336

Epoch 19/35

5514/5514 - 12s - loss: 0.1101 - accuracy: 0.9536 - val\_loss: 0.1738 - val\_accuracy: 0.9374

Epoch 20/35

5514/5514 - 12s - loss: 0.0991 - accuracy: 0.9530 - val\_loss: 0.2323 - val\_accuracy: 0.9260

Epoch 21/35

5514/5514 - 12s - loss: 0.0956 - accuracy: 0.9583 - val\_loss: 0.2394 - val\_accuracy: 0.9380

Epoch 00021: early stopping



show\_train\_history(train\_history,n\_epochs,'accuracy','val\_accuracy')

show\_train\_history(train\_history,n\_epochs,'loss','val\_loss')

In [14]:



*# define function for creating labeled confusion matrix for multiclass identification*

**def** labeled\_confusion\_mat(true\_vector, pred\_vector, labels):

confusion\_matrix\_labeled **=** pd.DataFrame(confusion\_matrix(true\_vector, pred\_vector))

confusion\_matrix\_labeled.columns **=** list(labels)

confusion\_matrix\_labeled.index **=** list(labels)

**return** confusion\_matrix\_labeled

In [79]:



**from** sklearn.metrics **import** classification\_report

**from** sklearn.metrics **import** confusion\_matrix

​

*# evaluate RNN model on test set using .predict() and then revert to integer class label encoding*

y\_test\_pred\_encoded **=** model.predict(X\_test\_scaled)

y\_test\_pred **=** np.argmax(y\_test\_pred\_encoded, axis **=**1)

y\_test\_act **=** np.argmax(y\_test, axis **=**1)

​

labels **=** ['LAY', 'SIT', 'STA', 'WALK', 'WALK\_D', 'WALK\_U']

print('Confusion matrix for RNN evaluation on training set:\n')

print(labeled\_confusion\_mat(y\_test\_act, y\_test\_pred, labels))

print('\nClassification report for RNN evaluation on training set:\n')

print(classification\_report(y\_test\_act, y\_test\_pred, target\_names**=**labels))

Confusion matrix for RNN evaluation on training set:

LAY SIT STA WALK WALK\_D WALK\_U

LAY 463 7 26 0 0 0

SIT 2 451 16 2 0 0

STA 0 0 420 0 0 0

WALK 0 21 0 408 57 5

WALK\_D 0 2 0 65 465 0

WALK\_U 0 19 0 0 0 518

Classification report for RNN evaluation on training set:

precision recall f1-score support

LAY 1.00 0.93 0.96 496

SIT 0.90 0.96 0.93 471

STA 0.91 1.00 0.95 420

WALK 0.86 0.83 0.84 491

WALK\_D 0.89 0.87 0.88 532

WALK\_U 0.99 0.96 0.98 537

accuracy 0.92 2947

macro avg 0.92 0.93 0.92 2947

weighted avg 0.93 0.92 0.92 2947

Reviewing the results, an overall accuracy of 0.92 is observed. The most often confused classes are WALK/WALK DOWNSTAIRS, while the LAY and WALK UPSTAIRS classes are those with the highest classification accuracy overall.

Now that an RNN model with a single LSTM layer has been evaluated, additional variations on this model may be tested for potential evaluation improvements. In particular, combining the LSTM with a convolutional neural network architecture is promising, as the sequences of 128 time steps can be subsampled and explored through convolution, a powerful pattern finding tool.

**CNN LSTM Model**



The CNN LSTM model is often implemented in machine learning tasks with input that has both temporal and spatial characteristics, such as sequences of images (video), or sequences of words/sentences in text. The CNN-LSTM architecture uses CNN layer(s) for feature extraction on the raw input data and uses subsequent LSTM layer(s) to support sequence prediction.

​

In order to create input for the initial CNN convolutional layer, the training data array is reshaped such that each time window of 128 samples is broken into 4 blocks or "steps" of 32 samples each, across each of the 9 features. The entire CNN model is wrapped in a TimeDistributed layer to allow the model to read in each of the four steps in the window. The features extracted by the CNN are then flattened and used as input to the LSTM model, which then extracts its own set of temporal features and then returns a final class prediction through the output layer.

​

Again, the model architecture will be defined in a function and a grid search used to evaluate hyperparameter combinations to optimize the model. Then the model with the best parameter combination will be fit to the training dataset and evaluated on the test dataset.

In [50]:



*# Use scikit-learn to grid search the network hyperparameters*

**from** tensorflow.keras.layers **import** Flatten

**from** tensorflow.keras.layers **import** TimeDistributed

**from** tensorflow.keras.layers **import** Conv1D

**from** tensorflow.keras.layers **import** MaxPooling1D

​

*# Define key array lengths*

n\_timesteps, n\_features, n\_outputs **=** X\_train.shape[1], X\_train.shape[2], y\_train.shape[1]

​

*# reshape data into time steps of sub-sequences*

n\_steps, n\_length **=** 4, 32

X\_train\_reshaped **=** X\_train\_scaled.reshape((X\_train.shape[0], n\_steps, n\_length, n\_features))

X\_test\_reshaped **=** X\_test\_scaled.reshape((X\_test.shape[0], n\_steps, n\_length, n\_features))

​

*# Function to create model, required for KerasClassifier*

**def** create\_CNN\_LSTM\_model(neurons**=**100, dropout**=**0.5, batch\_size**=**64):

​

*# define model*

model **=** Sequential()

model.add(TimeDistributed(Conv1D(filters**=**64, kernel\_size**=**3, activation**=**'relu'), input\_shape**=**(**None**,n\_length,n\_features)))

model.add(TimeDistributed(Conv1D(filters**=**64, kernel\_size**=**3, activation**=**'relu')))

model.add(TimeDistributed(Dropout(dropout)))

model.add(TimeDistributed(MaxPooling1D(pool\_size**=**2)))

model.add(TimeDistributed(Flatten()))

model.add(LSTM(neurons))

model.add(Dropout(dropout))

model.add(Dense(neurons, kernel\_initializer**=**'glorot\_uniform', activation**=**'relu'))

model.add(Dense(n\_outputs, activation**=**'softmax'))

model.compile(loss**=**'categorical\_crossentropy', optimizer**=**'adam', metrics**=**['accuracy'])

**return** model

In [44]:



*# create model with early stopping monitor (if model convergence reached before n\_epochs)*

n\_epochs **=** 35

model **=** KerasClassifier(build\_fn**=**create\_CNN\_LSTM\_model, verbose**=**0, epochs**=**n\_epochs)

es **=** EarlyStopping(monitor**=**'loss', mode**=**'min', verbose**=**1, patience**=**5)

​

*# define the grid search parameters*

neurons **=** [50, 100, 200]

dropout **=** [0.1,0.3,0.5]

batch\_size **=** [32, 64, 128]

​

*# neurons = [100]*

*# dropout = [0.5]*

*# batch\_size = [64]*

*#filters = [64]*

*#kernel\_size = [3]*

param\_grid **=** dict(neurons**=**neurons,

dropout**=**dropout,

batch\_size**=**batch\_size)

*#filters = filters,*

*#kernel\_size = kernel\_size)*

​

*# run grid search and report results and early stopping*

*# grid = GridSearchCV(estimator=model, param\_grid=param\_grid, n\_jobs=-1, cv=3, verbose=1)*

grid **=** GridSearchCV(estimator**=**model, param\_grid**=**param\_grid, cv**=**3, verbose**=**1, n\_jobs**=**1)

In [19]:



grid\_result **=** grid.fit(X\_train\_reshaped, y\_train, callbacks**=**[es])

Fitting 3 folds for each of 27 candidates, totalling 81 fits

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

Epoch 00026: early stopping

Epoch 00031: early stopping

Epoch 00030: early stopping

Epoch 00017: early stopping

Epoch 00035: early stopping

Epoch 00022: early stopping

Epoch 00021: early stopping

Epoch 00023: early stopping

Epoch 00030: early stopping

Epoch 00028: early stopping

Epoch 00028: early stopping

Epoch 00029: early stopping

Epoch 00027: early stopping

Epoch 00030: early stopping

Epoch 00023: early stopping

Epoch 00020: early stopping

Epoch 00034: early stopping

Epoch 00025: early stopping

Epoch 00025: early stopping

Epoch 00033: early stopping

Epoch 00033: early stopping

Epoch 00034: early stopping

Epoch 00033: early stopping

Epoch 00032: early stopping

Epoch 00019: early stopping

Epoch 00026: early stopping

Epoch 00033: early stopping

Epoch 00028: early stopping

Epoch 00029: early stopping

Epoch 00026: early stopping

[Parallel(n\_jobs=1)]: Done 81 out of 81 | elapsed: 76.7min finished

In [20]:



*# summarize results*

print("Best: %f using %s" **%** (grid\_result.best\_score\_, grid\_result.best\_params\_))

means **=** grid\_result.cv\_results\_['mean\_test\_score']

stds **=** grid\_result.cv\_results\_['std\_test\_score']

params **=** grid\_result.cv\_results\_['params']

**for** mean, stdev, param **in** zip(means, stds, params):

print("%f (%f) with: %r" **%** (mean, stdev, param))

Best: 0.940969 using {'batch\_size': 64, 'dropout': 0.1, 'neurons': 200}

0.922063 (0.024809) with: {'batch\_size': 32, 'dropout': 0.1, 'neurons': 50}

0.929545 (0.024330) with: {'batch\_size': 32, 'dropout': 0.1, 'neurons': 100}

0.929817 (0.026640) with: {'batch\_size': 32, 'dropout': 0.1, 'neurons': 200}

0.928865 (0.021968) with: {'batch\_size': 32, 'dropout': 0.3, 'neurons': 50}

0.912134 (0.030992) with: {'batch\_size': 32, 'dropout': 0.3, 'neurons': 100}

0.932808 (0.012928) with: {'batch\_size': 32, 'dropout': 0.3, 'neurons': 200}

0.926009 (0.018422) with: {'batch\_size': 32, 'dropout': 0.5, 'neurons': 50}

0.937432 (0.010236) with: {'batch\_size': 32, 'dropout': 0.5, 'neurons': 100}

0.937160 (0.014038) with: {'batch\_size': 32, 'dropout': 0.5, 'neurons': 200}

0.930497 (0.012188) with: {'batch\_size': 64, 'dropout': 0.1, 'neurons': 50}

0.920702 (0.008562) with: {'batch\_size': 64, 'dropout': 0.1, 'neurons': 100}

0.940969 (0.005572) with: {'batch\_size': 64, 'dropout': 0.1, 'neurons': 200}

0.929134 (0.010310) with: {'batch\_size': 64, 'dropout': 0.3, 'neurons': 50}

0.930359 (0.011486) with: {'batch\_size': 64, 'dropout': 0.3, 'neurons': 100}

0.933760 (0.008913) with: {'batch\_size': 64, 'dropout': 0.3, 'neurons': 200}

0.901793 (0.030657) with: {'batch\_size': 64, 'dropout': 0.5, 'neurons': 50}

0.936617 (0.009206) with: {'batch\_size': 64, 'dropout': 0.5, 'neurons': 100}

0.929679 (0.013159) with: {'batch\_size': 64, 'dropout': 0.5, 'neurons': 200}

0.922062 (0.016666) with: {'batch\_size': 128, 'dropout': 0.1, 'neurons': 50}

0.918525 (0.019501) with: {'batch\_size': 128, 'dropout': 0.1, 'neurons': 100}

0.920973 (0.024337) with: {'batch\_size': 128, 'dropout': 0.1, 'neurons': 200}

0.934031 (0.010590) with: {'batch\_size': 128, 'dropout': 0.3, 'neurons': 50}

0.920293 (0.017813) with: {'batch\_size': 128, 'dropout': 0.3, 'neurons': 100}

0.920567 (0.027558) with: {'batch\_size': 128, 'dropout': 0.3, 'neurons': 200}

0.909553 (0.023866) with: {'batch\_size': 128, 'dropout': 0.5, 'neurons': 50}

0.931449 (0.023420) with: {'batch\_size': 128, 'dropout': 0.5, 'neurons': 100}

0.922879 (0.022255) with: {'batch\_size': 128, 'dropout': 0.5, 'neurons': 200}

In [51]:



*# initialize a model with the best parameters from the grid search*

start\_time **=** time.time()

model **=** create\_CNN\_LSTM\_model(neurons**=**200,

dropout**=**0.1,

batch\_size**=**64)

​

*# model = create\_CNN\_LSTM\_model(neurons=grid\_result.best\_params\_['neurons'],*

*# dropout=grid\_result.best\_params\_['dropout'],*

*# batch\_size=grid\_result.best\_params\_['batch\_size'])*

​

*# fit the model and output the training history so it may be plotted*

train\_history **=** model.fit(x**=**X\_train\_reshaped, y**=**y\_train,

validation\_split**=**0.25,

epochs**=**n\_epochs,

*#batch\_size=grid\_result.best\_params\_['batch\_size'],*

batch\_size**=**64,

verbose**=**2,

callbacks**=**[es])

​

print("--- Training executed in %s seconds ---" **%** (time.time() **-** start\_time))

Train on 5514 samples, validate on 1838 samples

Epoch 1/35

5514/5514 - 4s - loss: 0.4546 - accuracy: 0.8063 - val\_loss: 0.2248 - val\_accuracy: 0.9222

Epoch 2/35

5514/5514 - 3s - loss: 0.1398 - accuracy: 0.9441 - val\_loss: 0.2228 - val\_accuracy: 0.9396

Epoch 3/35

5514/5514 - 2s - loss: 0.0908 - accuracy: 0.9581 - val\_loss: 0.2503 - val\_accuracy: 0.9429

Epoch 4/35

5514/5514 - 2s - loss: 0.0826 - accuracy: 0.9583 - val\_loss: 0.2856 - val\_accuracy: 0.9298

Epoch 5/35

5514/5514 - 3s - loss: 0.0691 - accuracy: 0.9652 - val\_loss: 0.3642 - val\_accuracy: 0.9314

Epoch 6/35

5514/5514 - 3s - loss: 0.0635 - accuracy: 0.9664 - val\_loss: 0.2746 - val\_accuracy: 0.9157

Epoch 7/35

5514/5514 - 3s - loss: 0.0687 - accuracy: 0.9621 - val\_loss: 0.3561 - val\_accuracy: 0.9342

Epoch 8/35

5514/5514 - 3s - loss: 0.0590 - accuracy: 0.9675 - val\_loss: 0.3673 - val\_accuracy: 0.9412

Epoch 9/35

5514/5514 - 3s - loss: 0.0568 - accuracy: 0.9690 - val\_loss: 0.4231 - val\_accuracy: 0.9478

Epoch 10/35

5514/5514 - 2s - loss: 0.0611 - accuracy: 0.9686 - val\_loss: 0.3589 - val\_accuracy: 0.9195

Epoch 11/35

5514/5514 - 3s - loss: 0.0547 - accuracy: 0.9703 - val\_loss: 0.3132 - val\_accuracy: 0.9140

Epoch 12/35

5514/5514 - 2s - loss: 0.0476 - accuracy: 0.9742 - val\_loss: 0.4000 - val\_accuracy: 0.9478

Epoch 13/35

5514/5514 - 2s - loss: 0.0616 - accuracy: 0.9724 - val\_loss: 0.5733 - val\_accuracy: 0.8972

Epoch 14/35

5514/5514 - 3s - loss: 0.0541 - accuracy: 0.9737 - val\_loss: 0.3678 - val\_accuracy: 0.9124

Epoch 15/35

5514/5514 - 3s - loss: 0.0470 - accuracy: 0.9746 - val\_loss: 0.4082 - val\_accuracy: 0.9396

Epoch 16/35

5514/5514 - 3s - loss: 0.0423 - accuracy: 0.9781 - val\_loss: 0.3579 - val\_accuracy: 0.9309

Epoch 17/35

5514/5514 - 3s - loss: 0.0346 - accuracy: 0.9820 - val\_loss: 0.5335 - val\_accuracy: 0.9010

Epoch 18/35

5514/5514 - 2s - loss: 0.0474 - accuracy: 0.9773 - val\_loss: 0.3485 - val\_accuracy: 0.9347

Epoch 19/35

5514/5514 - 2s - loss: 0.0400 - accuracy: 0.9786 - val\_loss: 0.4130 - val\_accuracy: 0.9206

Epoch 20/35

5514/5514 - 3s - loss: 0.0300 - accuracy: 0.9824 - val\_loss: 0.4543 - val\_accuracy: 0.9304

Epoch 21/35

5514/5514 - 3s - loss: 0.0352 - accuracy: 0.9815 - val\_loss: 0.3455 - val\_accuracy: 0.9489

Epoch 22/35

5514/5514 - 3s - loss: 0.0365 - accuracy: 0.9799 - val\_loss: 0.4853 - val\_accuracy: 0.9162

Epoch 23/35

5514/5514 - 2s - loss: 0.0337 - accuracy: 0.9837 - val\_loss: 0.4930 - val\_accuracy: 0.9396

Epoch 24/35

5514/5514 - 3s - loss: 0.0511 - accuracy: 0.9802 - val\_loss: 0.4639 - val\_accuracy: 0.9091

Epoch 25/35

5514/5514 - 3s - loss: 0.0316 - accuracy: 0.9853 - val\_loss: 0.4632 - val\_accuracy: 0.9271

Epoch 00025: early stopping

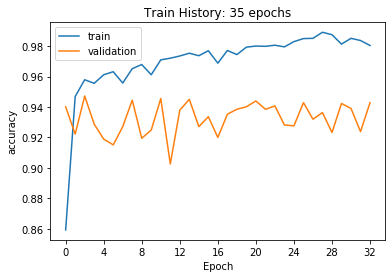
--- Executed in 64.88555431365967 seconds ---

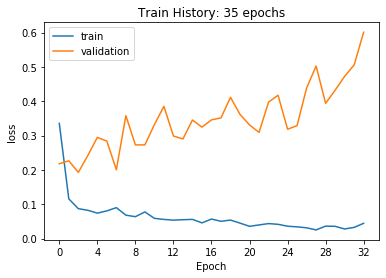
In [48]:



show\_train\_history(train\_history,n\_epochs,'accuracy','val\_accuracy')

show\_train\_history(train\_history,n\_epochs,'loss','val\_loss')





This outcome for the training history is challenging to understand at first, since normally an increase in validation loss would be expected to be accompanied by a decrease in validation accuracy. However, that is not the case here as the accuracy stays relatively constant but the loss increases significantly.

A possible explanation:

Cross entropy is not a bounded loss, meaning that a few "very" wrong predictions can potentially make the loss grow very quickly. It is possible that there are a few outliers that are classified extremely badly and that are making the loss explode, but the model is still learning on the rest of the dataset and continuing to correctly predict the other samples consistently.

The growing loss could be a combination of 2 competing phenomena:

* Some examples with borderline predictions get predicted better and so their output class changes (e.g. a SITTING sample predicted at 0.4 to be SITTING and 0.6 to STANDING in the next epoch gets predicted at 0.4 to be STANDING and 0.6 to be SITTING). In this case, accuracy increases while loss decreases.
* Some examples with poor predictions keep getting worse (e.g. a SITTING sample predicted at 0.8 to be to be STANDING becomes predicted at 0.9 to be STANDING) and/or, potentially more probable for multi-class problems, some examples with very good predictions get a little worse (e.g. a SITTING sample predicted at 0.9 to be SITTING becomes predicted at 0.8 to be SITTING). In this case, loss increases while accuracy stays the same.

If the second case occurs on lots of examples (e.g. for a specific class which is not well captured by the model, such as SITTING or WALKING classes) and cannot be compensated enough by the first case, then the loss will continually increase.

Several attempts at creating a converging or decreasing loss were attempted, with no success for this particular model architecture:

* increasing the number of epochs
* decreasing the batch size
* decreasing and increasing the "step" (subsequence) length

In [49]:



**from** sklearn.metrics **import** classification\_report

**from** sklearn.metrics **import** confusion\_matrix

​

*# evaluate model on test set using .predict() and then revert to integer class label encoding*

y\_test\_pred\_encoded **=** model.predict(X\_test\_reshaped)

y\_test\_pred **=** np.argmax(y\_test\_pred\_encoded, axis **=**1)

y\_test\_act **=** np.argmax(y\_test, axis **=**1)

​

labels **=** ['LAY', 'SIT', 'STA', 'WALK', 'WALK\_D', 'WALK\_U']

print('Confusion matrix for CNN LSTM evaluation on test set:\n')

print(labeled\_confusion\_mat(y\_test\_act, y\_test\_pred, labels))

print('\nClassification report for CNN LSTM evaluation on test set:\n')

print(classification\_report(y\_test\_act, y\_test\_pred, target\_names**=**labels))

Confusion matrix for CNN LSTM evaluation on test set:

LAY SIT STA WALK WALK\_D WALK\_U

LAY 470 1 25 0 0 0

SIT 2 443 25 1 0 0

STA 7 11 401 1 0 0

WALK 0 9 0 406 75 1

WALK\_D 0 0 0 70 462 0

WALK\_U 0 18 0 0 0 519

Classification report for CNN LSTM evaluation on test set:

precision recall f1-score support

LAY 0.98 0.95 0.96 496

SIT 0.92 0.94 0.93 471

STA 0.89 0.95 0.92 420

WALK 0.85 0.83 0.84 491

WALK\_D 0.86 0.87 0.86 532

WALK\_U 1.00 0.97 0.98 537

accuracy 0.92 2947

macro avg 0.92 0.92 0.92 2947

weighted avg 0.92 0.92 0.92 2947

**ConvLSTM Model**

The ConvLSTM is similar in function to the CNN-LSTM. In ConvLSTM, the matrix multiplication calculation of the input with the LSTM cell replaced by a convolution operation; the convolution is essentially embedded in the architecture. In contrast, CNN-LSTM architecture concatenates the CNN and LSTM architectures together externally.

Again, the model architecture will be defined in a function and a grid search used to evaluate hyperparameter combinations to optimize the model. Then the model with the best parameter combination will be fit to the training dataset and evaluated on the test dataset.

In [52]:



*# Use scikit-learn to grid search the network hyperparameters*

**from** tensorflow.keras.layers **import** ConvLSTM2D

​

*# Define key array lengths*

n\_timesteps, n\_features, n\_outputs **=** X\_train.shape[1], X\_train.shape[2], y\_train.shape[1]

*# reshape data into time steps of sub-sequences*

n\_steps, n\_length **=** 4, 32

X\_train\_reshaped **=** X\_train\_scaled.reshape((X\_train.shape[0], n\_steps, 1, n\_length, n\_features))

X\_test\_reshaped **=** X\_test\_scaled.reshape((X\_test.shape[0], n\_steps, 1, n\_length, n\_features))

​

*# Function to create model, required for KerasClassifier*

**def** create\_ConvLSTM\_model(neurons**=**100, dropout**=**0.5, batch\_size**=**64):

*# define model*

model **=** Sequential()

model.add(ConvLSTM2D(filters**=**64, kernel\_size**=**(1,3), activation**=**'relu', input\_shape**=**(n\_steps, 1, n\_length, n\_features)))

model.add(Dropout(dropout))

model.add(Flatten())

model.add(Dense(neurons, kernel\_initializer**=**'glorot\_uniform', activation**=**'relu'))

model.add(Dense(n\_outputs, activation**=**'softmax'))

model.compile(loss**=**'categorical\_crossentropy', optimizer**=**'adam', metrics**=**['accuracy'])

**return** model

In [53]:



*# create model with early stopping monitor (if model convergence reached before n\_epochs)*

n\_epochs **=** 35

model **=** KerasClassifier(build\_fn**=**create\_ConvLSTM\_model, verbose**=**0, epochs**=**n\_epochs)

es **=** EarlyStopping(monitor**=**'loss', mode**=**'min', verbose**=**1, patience**=**5)

​

*# define the grid search parameters*

neurons **=** [50, 100, 200]

dropout **=** [0.1,0.3,0.5]

batch\_size **=** [32, 64, 128]

​

*# neurons = [100]*

*# dropout = [0.5]*

*# batch\_size = [64]*

*# #filters = [64]*

*# #kernel\_size = [3]*

param\_grid **=** dict(neurons**=**neurons,

dropout**=**dropout,

batch\_size**=**batch\_size)

*#filters = filters,*

*#kernel\_size = kernel\_size)*

​

*# run grid search and report results and early stopping*

grid **=** GridSearchCV(estimator**=**model, param\_grid**=**param\_grid, cv**=**3, verbose**=**1, n\_jobs**=**1)

In [54]:



grid\_result **=** grid.fit(X\_train\_reshaped, y\_train, callbacks**=**[es])

Fitting 3 folds for each of 27 candidates, totalling 81 fits

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

Epoch 00021: early stopping

Epoch 00029: early stopping

Epoch 00035: early stopping

Epoch 00033: early stopping

Epoch 00023: early stopping

Epoch 00020: early stopping

Epoch 00029: early stopping

Epoch 00033: early stopping

Epoch 00027: early stopping

Epoch 00027: early stopping

Epoch 00032: early stopping

Epoch 00030: early stopping

Epoch 00035: early stopping

Epoch 00022: early stopping

Epoch 00034: early stopping

Epoch 00025: early stopping

Epoch 00022: early stopping

Epoch 00027: early stopping

Epoch 00035: early stopping

Epoch 00033: early stopping

Epoch 00032: early stopping

[Parallel(n\_jobs=1)]: Done 81 out of 81 | elapsed: 152.6min finished

In [55]:



*# summarize results*

print("Best: %f using %s" **%** (grid\_result.best\_score\_, grid\_result.best\_params\_))

means **=** grid\_result.cv\_results\_['mean\_test\_score']

stds **=** grid\_result.cv\_results\_['std\_test\_score']

params **=** grid\_result.cv\_results\_['params']

**for** mean, stdev, param **in** zip(means, stds, params):

print("%f (%f) with: %r" **%** (mean, stdev, param))

Best: 0.937432 using {'batch\_size': 32, 'dropout': 0.1, 'neurons': 200}

0.918115 (0.017207) with: {'batch\_size': 32, 'dropout': 0.1, 'neurons': 50}

0.917983 (0.028706) with: {'batch\_size': 32, 'dropout': 0.1, 'neurons': 100}

0.937432 (0.007025) with: {'batch\_size': 32, 'dropout': 0.1, 'neurons': 200}

0.932672 (0.010798) with: {'batch\_size': 32, 'dropout': 0.3, 'neurons': 50}

0.924646 (0.020988) with: {'batch\_size': 32, 'dropout': 0.3, 'neurons': 100}

0.929407 (0.016085) with: {'batch\_size': 32, 'dropout': 0.3, 'neurons': 200}

0.922333 (0.016311) with: {'batch\_size': 32, 'dropout': 0.5, 'neurons': 50}

0.922877 (0.018884) with: {'batch\_size': 32, 'dropout': 0.5, 'neurons': 100}

0.935256 (0.009173) with: {'batch\_size': 32, 'dropout': 0.5, 'neurons': 200}

0.916076 (0.023411) with: {'batch\_size': 64, 'dropout': 0.1, 'neurons': 50}

0.922467 (0.019648) with: {'batch\_size': 64, 'dropout': 0.1, 'neurons': 100}

0.924780 (0.015674) with: {'batch\_size': 64, 'dropout': 0.1, 'neurons': 200}

0.924099 (0.020997) with: {'batch\_size': 64, 'dropout': 0.3, 'neurons': 50}

0.907101 (0.004509) with: {'batch\_size': 64, 'dropout': 0.3, 'neurons': 100}

0.927776 (0.015844) with: {'batch\_size': 64, 'dropout': 0.3, 'neurons': 200}

0.928999 (0.025345) with: {'batch\_size': 64, 'dropout': 0.5, 'neurons': 50}

0.928589 (0.017324) with: {'batch\_size': 64, 'dropout': 0.5, 'neurons': 100}

0.927638 (0.016309) with: {'batch\_size': 64, 'dropout': 0.5, 'neurons': 200}

0.927910 (0.012120) with: {'batch\_size': 128, 'dropout': 0.1, 'neurons': 50}

0.918253 (0.007127) with: {'batch\_size': 128, 'dropout': 0.1, 'neurons': 100}

0.928319 (0.014825) with: {'batch\_size': 128, 'dropout': 0.1, 'neurons': 200}

0.922605 (0.016515) with: {'batch\_size': 128, 'dropout': 0.3, 'neurons': 50}

0.921652 (0.017639) with: {'batch\_size': 128, 'dropout': 0.3, 'neurons': 100}

0.920973 (0.023527) with: {'batch\_size': 128, 'dropout': 0.3, 'neurons': 200}

0.911045 (0.027496) with: {'batch\_size': 128, 'dropout': 0.5, 'neurons': 50}

0.915805 (0.007782) with: {'batch\_size': 128, 'dropout': 0.5, 'neurons': 100}

0.921245 (0.018897) with: {'batch\_size': 128, 'dropout': 0.5, 'neurons': 200}

In [56]:



*# initialize a model with the best parameters from the grid search*

start\_time **=** time.time()

*# model = create\_ConvLSTM\_model(neurons=200,*

*# dropout=0.1,*

*# batch\_size=64)*

​

model **=** create\_ConvLSTM\_model(neurons**=**grid\_result.best\_params\_['neurons'],

dropout**=**grid\_result.best\_params\_['dropout'],

batch\_size**=**grid\_result.best\_params\_['batch\_size'])

​

*# fit the model and output the training history so it may be plotted*

train\_history **=** model.fit(x**=**X\_train\_reshaped, y**=**y\_train,

validation\_split**=**0.25,

epochs**=**n\_epochs,

*#batch\_size=grid\_result.best\_params\_['batch\_size'],*

batch\_size**=**64,

verbose**=**2,

callbacks**=**[es])

​

print("--- Training executed in %s seconds ---" **%** (time.time() **-** start\_time))

Train on 5514 samples, validate on 1838 samples

Epoch 1/35

5514/5514 - 7s - loss: 0.4284 - accuracy: 0.8333 - val\_loss: 0.1990 - val\_accuracy: 0.9407

Epoch 2/35

5514/5514 - 5s - loss: 0.1201 - accuracy: 0.9485 - val\_loss: 0.2069 - val\_accuracy: 0.9396

Epoch 3/35

5514/5514 - 5s - loss: 0.1129 - accuracy: 0.9487 - val\_loss: 0.1985 - val\_accuracy: 0.9440

Epoch 4/35

5514/5514 - 5s - loss: 0.0924 - accuracy: 0.9574 - val\_loss: 0.2214 - val\_accuracy: 0.9336

Epoch 5/35

5514/5514 - 5s - loss: 0.0837 - accuracy: 0.9594 - val\_loss: 0.2437 - val\_accuracy: 0.9450

Epoch 6/35

5514/5514 - 5s - loss: 0.0977 - accuracy: 0.9588 - val\_loss: 0.2674 - val\_accuracy: 0.9244

Epoch 7/35

5514/5514 - 5s - loss: 0.0852 - accuracy: 0.9608 - val\_loss: 0.2696 - val\_accuracy: 0.9412

Epoch 8/35

5514/5514 - 5s - loss: 0.0729 - accuracy: 0.9648 - val\_loss: 0.3023 - val\_accuracy: 0.9445

Epoch 9/35

5514/5514 - 5s - loss: 0.0690 - accuracy: 0.9677 - val\_loss: 0.3565 - val\_accuracy: 0.9249

Epoch 10/35

5514/5514 - 5s - loss: 0.1028 - accuracy: 0.9583 - val\_loss: 0.2836 - val\_accuracy: 0.9249

Epoch 11/35

5514/5514 - 5s - loss: 0.0668 - accuracy: 0.9657 - val\_loss: 0.3911 - val\_accuracy: 0.9211

Epoch 12/35

5514/5514 - 5s - loss: 0.0590 - accuracy: 0.9686 - val\_loss: 0.4069 - val\_accuracy: 0.9168

Epoch 13/35

5514/5514 - 5s - loss: 0.0607 - accuracy: 0.9701 - val\_loss: 0.3119 - val\_accuracy: 0.9189

Epoch 14/35

5514/5514 - 5s - loss: 0.0551 - accuracy: 0.9737 - val\_loss: 0.4109 - val\_accuracy: 0.9309

Epoch 15/35

5514/5514 - 5s - loss: 0.0524 - accuracy: 0.9704 - val\_loss: 0.4304 - val\_accuracy: 0.9320

Epoch 16/35

5514/5514 - 5s - loss: 0.0480 - accuracy: 0.9750 - val\_loss: 0.4397 - val\_accuracy: 0.9151

Epoch 17/35

5514/5514 - 5s - loss: 0.0461 - accuracy: 0.9755 - val\_loss: 0.4475 - val\_accuracy: 0.9412

Epoch 18/35

5514/5514 - 5s - loss: 0.0447 - accuracy: 0.9788 - val\_loss: 0.4595 - val\_accuracy: 0.9407

Epoch 19/35

5514/5514 - 5s - loss: 0.0405 - accuracy: 0.9804 - val\_loss: 0.4650 - val\_accuracy: 0.9189

Epoch 20/35

5514/5514 - 5s - loss: 0.0526 - accuracy: 0.9791 - val\_loss: 0.3932 - val\_accuracy: 0.9211

Epoch 21/35

5514/5514 - 5s - loss: 0.0494 - accuracy: 0.9791 - val\_loss: 0.4559 - val\_accuracy: 0.9266

Epoch 22/35

5514/5514 - 5s - loss: 0.0394 - accuracy: 0.9797 - val\_loss: 0.5134 - val\_accuracy: 0.9287

Epoch 23/35

5514/5514 - 5s - loss: 0.0395 - accuracy: 0.9820 - val\_loss: 0.4455 - val\_accuracy: 0.9255

Epoch 24/35

5514/5514 - 5s - loss: 0.0340 - accuracy: 0.9837 - val\_loss: 0.5224 - val\_accuracy: 0.9331

Epoch 25/35

5514/5514 - 5s - loss: 0.0313 - accuracy: 0.9842 - val\_loss: 0.5084 - val\_accuracy: 0.9222

Epoch 26/35

5514/5514 - 5s - loss: 0.0291 - accuracy: 0.9864 - val\_loss: 0.5710 - val\_accuracy: 0.9304

Epoch 27/35

5514/5514 - 5s - loss: 0.0277 - accuracy: 0.9871 - val\_loss: 0.7086 - val\_accuracy: 0.9429

Epoch 28/35

5514/5514 - 5s - loss: 0.0247 - accuracy: 0.9904 - val\_loss: 0.6866 - val\_accuracy: 0.9347

Epoch 29/35

5514/5514 - 5s - loss: 0.0428 - accuracy: 0.9830 - val\_loss: 0.5907 - val\_accuracy: 0.9200

Epoch 30/35

5514/5514 - 5s - loss: 0.0556 - accuracy: 0.9810 - val\_loss: 0.5010 - val\_accuracy: 0.9353

Epoch 31/35

5514/5514 - 5s - loss: 0.0315 - accuracy: 0.9895 - val\_loss: 0.5361 - val\_accuracy: 0.9423

Epoch 32/35

5514/5514 - 5s - loss: 0.0297 - accuracy: 0.9868 - val\_loss: 0.5994 - val\_accuracy: 0.9440

Epoch 33/35

5514/5514 - 5s - loss: 0.0232 - accuracy: 0.9904 - val\_loss: 0.7224 - val\_accuracy: 0.9456

Epoch 34/35

5514/5514 - 5s - loss: 0.0189 - accuracy: 0.9927 - val\_loss: 0.7479 - val\_accuracy: 0.9429

Epoch 35/35

5514/5514 - 5s - loss: 0.0211 - accuracy: 0.9911 - val\_loss: 0.7308 - val\_accuracy: 0.9467

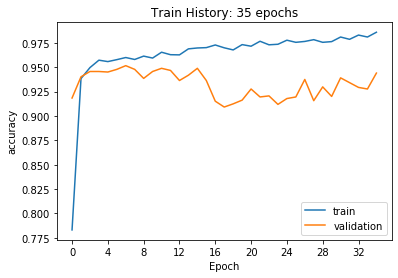
--- Training executed in 165.32012295722961 seconds ---

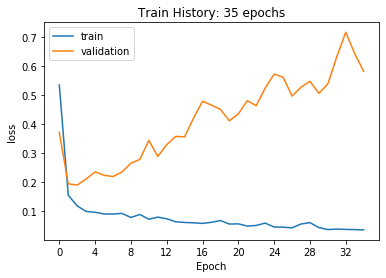
In [40]:



show\_train\_history(train\_history,n\_epochs,'accuracy','val\_accuracy')

show\_train\_history(train\_history,n\_epochs,'loss','val\_loss')





In [41]:



**from** sklearn.metrics **import** classification\_report

**from** sklearn.metrics **import** confusion\_matrix

​

*# evaluate model on test set using .predict() and then revert to integer class label encoding*

y\_test\_pred\_encoded **=** model.predict(X\_test\_reshaped)

y\_test\_pred **=** np.argmax(y\_test\_pred\_encoded, axis **=**1)

y\_test\_act **=** np.argmax(y\_test, axis **=**1)

​

labels **=** ['LAY', 'SIT', 'STA', 'WALK', 'WALK\_D', 'WALK\_U']

print('Confusion matrix for ConvLSTM evaluation on test set:\n')

print(labeled\_confusion\_mat(y\_test\_act, y\_test\_pred, labels))

print('\nClassification report for ConvLSTM evaluation on test set:\n')

print(classification\_report(y\_test\_act, y\_test\_pred, target\_names**=**labels))

Confusion matrix for ConvLSTM evaluation on test set:

LAY SIT STA WALK WALK\_D WALK\_U

LAY 464 5 27 0 0 0

SIT 4 433 34 0 0 0

STA 0 6 414 0 0 0

WALK 0 3 0 396 87 5

WALK\_D 1 0 0 56 475 0

WALK\_U 0 0 0 0 0 537

Classification report for ConvLSTM evaluation on test set:

precision recall f1-score support

LAY 0.99 0.94 0.96 496

SIT 0.97 0.92 0.94 471

STA 0.87 0.99 0.93 420

WALK 0.88 0.81 0.84 491

WALK\_D 0.85 0.89 0.87 532

WALK\_U 0.99 1.00 1.00 537

accuracy 0.92 2947

macro avg 0.92 0.92 0.92 2947

weighted avg 0.92 0.92 0.92 2947

Both the CNN-LSTM and ConvLSTM performed approximately the same, and not much better than the LSTM alone for this classification problem. None of the deep learning models implemented on the raw data were able to outperform the SVM baseline on the feature engineered data. However, their strong performance on the raw data is impressive given the knowledge required to perform the original feature engineering.