**Spotify Song Popularity Deep Learning Model**

*# Setup plotting*

import matplotlib.pyplot as plt

plt.style.use('seaborn-whitegrid')

*# Set Matplotlib defaults*

plt.rc('figure', autolayout=True)

plt.rc('axes', labelweight='bold', labelsize='large',

titleweight='bold', titlesize=18, titlepad=10)

plt.rc('animation', html='html5')

First load the Spotify dataset. Your task will be to predict the popularity of a song based on various audio features,

import pandas as pd

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.compose import make\_column\_transformer

from sklearn.model\_selection import GroupShuffleSplit

from tensorflow import keras

from tensorflow.keras import layers

from tensorflow.keras import callbacks

spotify = pd.read\_csv('../input/dl-course-data/spotify.csv')

X = spotify.copy().dropna()

y = X.pop('track\_popularity')

artists = X['track\_artist']

features\_num = ['danceability', 'energy', 'key', 'loudness', 'mode',

'speechiness', 'acousticness', 'instrumentalness',

'liveness', 'valence', 'tempo', 'duration\_ms']

features\_cat = ['playlist\_genre']

preprocessor = make\_column\_transformer(

(StandardScaler(), features\_num),

(OneHotEncoder(), features\_cat),

)

*# We'll do a "grouped" split to keep all of an artist's songs in one split or the other. This is to help prevent signal leakage.*

def group\_split(X, y, group, train\_size=0.75):

splitter = GroupShuffleSplit(train\_size=train\_size)

train, test = next(splitter.split(X, y, groups=group))

return (X.iloc[train], X.iloc[test], y.iloc[train], y.iloc[test])

X\_train, X\_valid, y\_train, y\_valid = group\_split(X, y, artists)

X\_train = preprocessor.fit\_transform(X\_train)

X\_valid = preprocessor.transform(X\_valid)

y\_train = y\_train / 100 *# popularity is on a scale 0-100, so this rescales to 0-1.*

y\_valid = y\_valid / 100

input\_shape = [X\_train.shape[1]]

print("Input shape: **{}**".format(input\_shape))

Let's start with the simplest network, a linear model. This model has low capacity.

Run this next cell without any changes to train a linear model on the Spotify dataset.

model = keras.Sequential([

layers.Dense(1, input\_shape=input\_shape),

])

model.compile(

optimizer='adam',

loss='mae',

)

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_valid, y\_valid),

batch\_size=512,

epochs=50,

verbose=0, *# suppress output since we'll plot the curves*

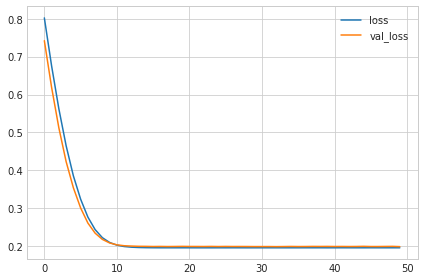
)

history\_df = pd.DataFrame(history.history)

history\_df.loc[0:, ['loss', 'val\_loss']].plot()

print("Minimum Validation Loss: **{:0.4f}**".format(history\_df['val\_loss'].min()));

Minimum Validation Loss: 0.1988

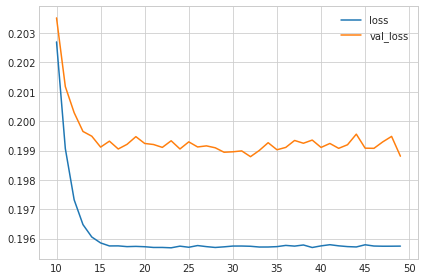


It's not uncommon for the curves to follow a "hockey stick" pattern like you see here. This makes the final part of training hard to see, so let's start at epoch 10 instead:

*# Start the plot at epoch 10*

history\_df.loc[10:, ['loss', 'val\_loss']].plot()

print("Minimum Validation Loss: **{:0.4f}**".format(history\_df['val\_loss'].min()));



1) Evaluate Baseline

The gap between these curves is quite small and the validation loss never increases, so it's more likely that the network is underfitting than overfitting. It would be worth experimenting with more capacity to see if that's the case.

Now let's add some capacity to our network. We'll add three hidden layers with 128 units each. Run the next cell to train the network and see the learning curves.

model = keras.Sequential([

layers.Dense(128, activation='relu', input\_shape=input\_shape),

layers.Dense(64, activation='relu'),

layers.Dense(1)

])

model.compile(

optimizer='adam',

loss='mae',

)

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_valid, y\_valid),

batch\_size=512,

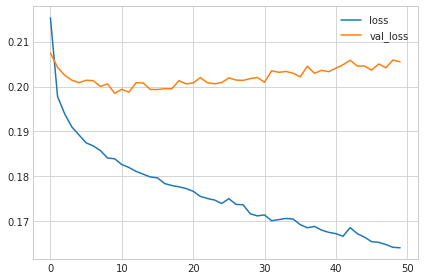
epochs=50,

)

history\_df = pd.DataFrame(history.history)

history\_df.loc[:, ['loss', 'val\_loss']].plot()

print("Minimum Validation Loss: **{:0.4f}**".format(history\_df['val\_loss'].min()));



# 2) Add Capacity

Now the validation loss begins to rise very early, while the training loss continues to decrease. This indicates that the network has begun to overfit. At this point, we would need to try something to prevent it, either by reducing the number of units or through a method like early stopping. (We'll see another in the next lesson!)

# 3) Define Early Stopping Callback

Now define an early stopping callback that waits 5 epochs (patience') for a change in validation loss of at least 0.001 (min\_delta) and keeps the weights with the best loss (restore\_best\_weights).

from tensorflow.keras import callbacks

*# YOUR CODE HERE: define an early stopping callback*

early\_stopping = callbacks.EarlyStopping(

min\_delta=0.001,

patience = 5,

restore\_best\_weights=True,

)

Now run this cell to train the model and get the learning curves. Notice the callbacks argument in model.fit.

model = keras.Sequential([

layers.Dense(128, activation='relu', input\_shape=input\_shape),

layers.Dense(64, activation='relu'),

layers.Dense(1)

])

model.compile(

optimizer='adam',

loss='mae',

)

history = model.fit(

X\_train, y\_train,

validation\_data=(X\_valid, y\_valid),

batch\_size=512,

epochs=50,

callbacks=[early\_stopping]

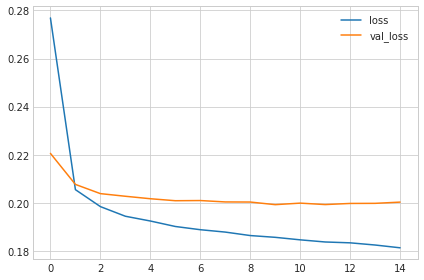
)

history\_df = pd.DataFrame(history.history)

history\_df.loc[:, ['loss', 'val\_loss']].plot()

print("Minimum Validation Loss: **{:0.4f}**".format(history\_df['val\_loss'].min()));

Minimum Validation Loss: 0.1993



# 4) Train and Interpret

Was this an improvement compared to training without early stopping?

The early stopping callback did stop the training once the network began overfitting. Moreover, by including restore\_best\_weights we still get to keep the model where validation loss was lowest.

If you like, try experimenting with patience and min\_delta to see what difference it might make.