This notebook is an exercise in the Intro to Deep Learning course. You can reference the tutorial at this link.

Introduction

In this exercise, you'll learn how to improve training outcomes by including an early stopping callback to prevent overfitting.

When you're ready, run this next cell to set everything up!

In [1]:

```
# 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')

# Setup feedback system
from learntools.core import binder
binder.bind(globals())
from learntools.deep_learning_intro.ex4 import *
```

First load the *Spotify* dataset. Your task will be to predict the popularity of a song based on various audio features, like `'tempo'`, `'danceability'`, and `'mode'`.

In [2]:

```
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))
```

```
Input shape: [18]
```

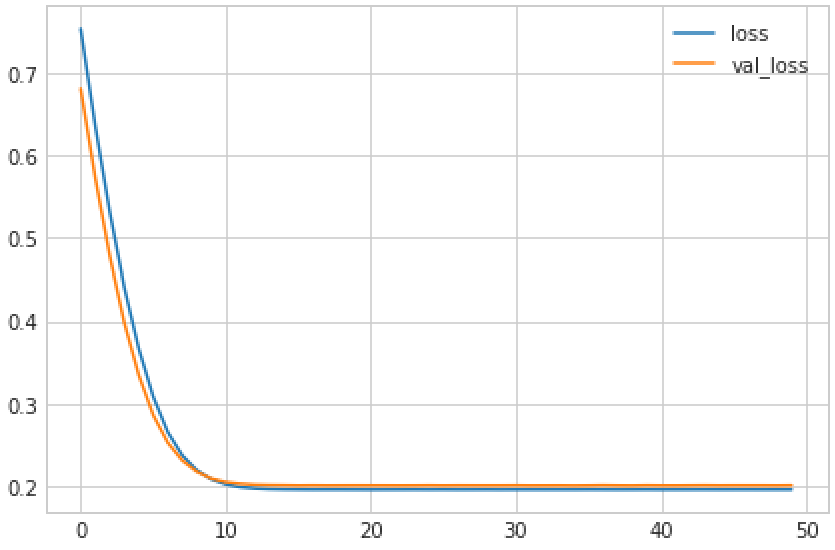
Let's start with the simplest network, a linear model. This model has low capacity.

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

In [3]:
```
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()));
```

```
2022-12-18 05:33:22.399276: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-18 05:33:22.491172: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-18 05:33:22.491936: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-18 05:33:22.493456: I tensorflow/core/platform/cpu_feature_guard.cc:142] This Te
nsorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the
following CPU instructions in performance-critical operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler fla
gs.
2022-12-18 05:33:22.493806: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-18 05:33:22.494750: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
```

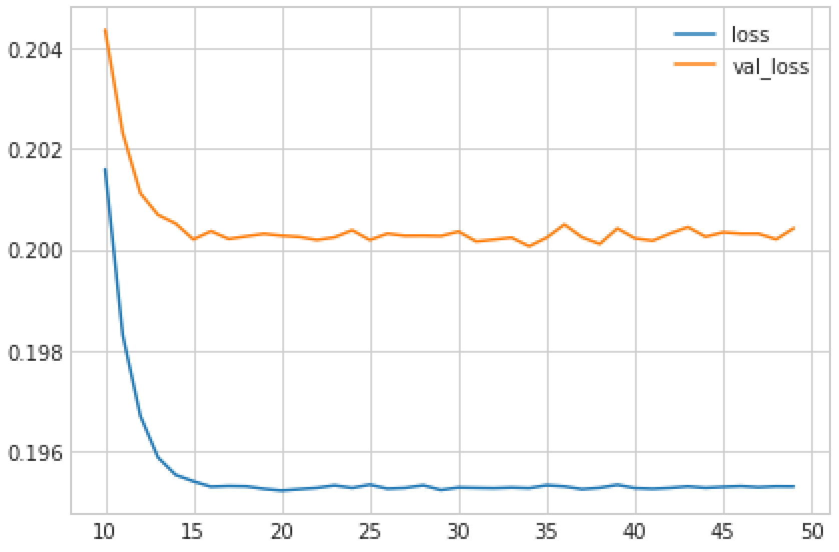
```
one NUMA node, so returning NUMA node zero
2022-12-18 05:33:22.495734: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-18 05:33:24.631735: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-18 05:33:24.632617: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-18 05:33:24.633299: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937]
successful NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero
2022-12-18 05:33:24.633887: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Cre
ated device /job:localhost/replica:0/task:0/device:GPU:0 with 15401 MB memory:  -> devic
e: 0, name: Tesla P100-PCIE-16GB, pci bus id: 0000:00:04.0, compute capability: 6.0
2022-12-18 05:33:25.096847: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:1
85] None of the MLIR Optimization Passes are enabled (registered 2)
Minimum Validation Loss: 0.2001
```



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:

In [4]:
```
# 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()));
```

```
Minimum Validation Loss: 0.2001
```



1) Evaluate Baseline

What do you think? Would you say this model is underfitting, overfitting, just right?

In [5]:
```
# View the solution (Run this cell to receive credit!)
q_1.check()
```

Correct:

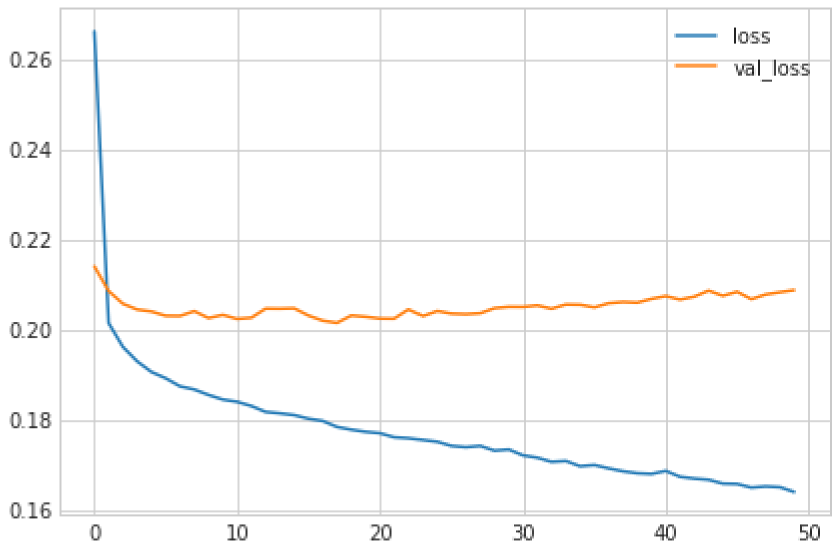
The gap between these curves is quite small and the validation loss never increases, so it's more likely that the network is underfitting than overfitting. It would be worth experimenting with more capacity to see if that's the case.

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.

In [6]:
```
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()));
```

```
Epoch 1/50
48/48 [==============================] - 1s 7ms/step - loss: 0.2660 - val_loss: 0.2139
Epoch 2/50
48/48 [==============================] - 0s 5ms/step - loss: 0.2014 - val_loss: 0.2085
Epoch 3/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1961 - val_loss: 0.2057
Epoch 4/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1929 - val_loss: 0.2044
Epoch 5/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1905 - val_loss: 0.2039
Epoch 6/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1891 - val_loss: 0.2030
Epoch 7/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1873 - val_loss: 0.2029
Epoch 8/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1866 - val_loss: 0.2040
Epoch 9/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1854 - val_loss: 0.2025
Epoch 10/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1844 - val_loss: 0.2032
Epoch 11/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1839 - val_loss: 0.2023
Epoch 12/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1830 - val_loss: 0.2025
Epoch 13/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1817 - val_loss: 0.2046
Epoch 14/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1813 - val_loss: 0.2046
Epoch 15/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1810 - val_loss: 0.2046
Epoch 16/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1802 - val_loss: 0.2030
Epoch 17/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1797 - val_loss: 0.2019
Epoch 18/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1783 - val_loss: 0.2014
Epoch 19/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1777 - val_loss: 0.2030
Epoch 20/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1773 - val_loss: 0.2028
Epoch 21/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1770 - val_loss: 0.2024
Epoch 22/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1760 - val_loss: 0.2023
Epoch 23/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1758 - val_loss: 0.2044
Epoch 24/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1754 - val_loss: 0.2029
Epoch 25/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1750 - val_loss: 0.2040
Epoch 26/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1741 - val_loss: 0.2035
Epoch 27/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1739 - val_loss: 0.2033
Epoch 28/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1741 - val_loss: 0.2035
Epoch 29/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1731 - val_loss: 0.2047
Epoch 30/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1733 - val_loss: 0.2050
```

```
Epoch 31/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1720 - val_loss: 0.2049
Epoch 32/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1715 - val_loss: 0.2053
Epoch 33/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1706 - val_loss: 0.2045
Epoch 34/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1708 - val_loss: 0.2055
Epoch 35/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1696 - val_loss: 0.2054
Epoch 36/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1699 - val_loss: 0.2049
Epoch 37/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1691 - val_loss: 0.2058
Epoch 38/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1685 - val_loss: 0.2060
Epoch 39/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1681 - val_loss: 0.2059
Epoch 40/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1679 - val_loss: 0.2067
Epoch 41/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1686 - val_loss: 0.2073
Epoch 42/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1673 - val_loss: 0.2066
Epoch 43/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1669 - val_loss: 0.2072
Epoch 44/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1666 - val_loss: 0.2086
Epoch 45/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1658 - val_loss: 0.2074
Epoch 46/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1657 - val_loss: 0.2083
Epoch 47/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1649 - val_loss: 0.2067
Epoch 48/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1651 - val_loss: 0.2077
Epoch 49/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1650 - val_loss: 0.2082
Epoch 50/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1639 - val_loss: 0.2087
Minimum Validation Loss: 0.2014
```



2) Add Capacity

What is your evaluation of these curves? Underfitting, overfitting, just right?

In [7]:
```
# View the solution (Run this cell to receive credit!)
q_2.check()
```

Correct:

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`).

In [8]:
```
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_weigh

# Check your answer
q_3.check()
```

Correct

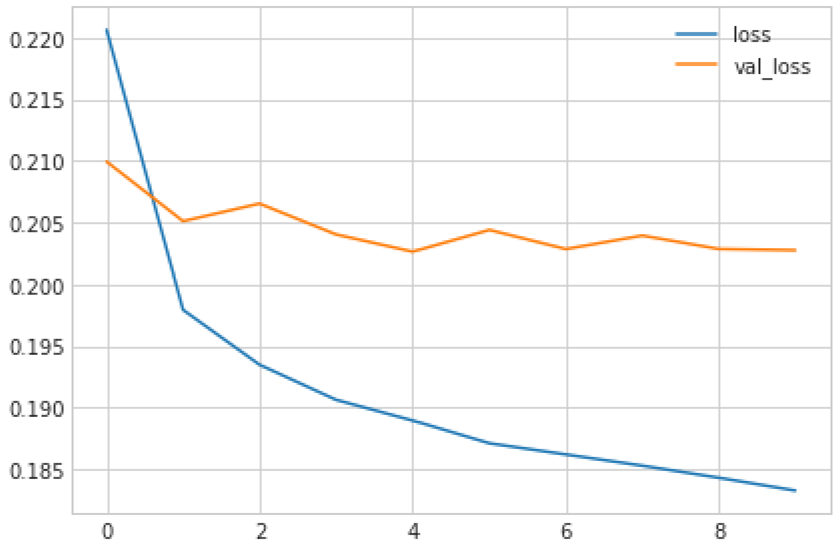
In [9]:
```
# Lines below will give you a hint or solution code
#q_3.hint()
#q_3.solution()
```

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

In [10]:
```
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()));
```

```
Epoch 1/50
48/48 [==============================] - 1s 7ms/step - loss: 0.2207 - val_loss: 0.2100
Epoch 2/50
48/48 [==============================] - 0s 4ms/step - loss: 0.1979 - val_loss: 0.2051
Epoch 3/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1935 - val_loss: 0.2065
Epoch 4/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1906 - val_loss: 0.2040
Epoch 5/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1890 - val_loss: 0.2026
Epoch 6/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1871 - val_loss: 0.2044
Epoch 7/50
48/48 [==============================] - 0s 2ms/step - loss: 0.1862 - val_loss: 0.2028
Epoch 8/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1853 - val_loss: 0.2039
Epoch 9/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1843 - val_loss: 0.2029
Epoch 10/50
48/48 [==============================] - 0s 3ms/step - loss: 0.1833 - val_loss: 0.2027
Minimum Validation Loss: 0.2026
```



4) Train and Interpret

Was this an improvement compared to training without early stopping?

In [11]:
```
# View the solution (Run this cell to receive credit!)
q_4.check()
```

Correct:

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.

Keep Going

Move on to **learn about a couple of special layers**: batch normalization and dropout.

Have questions or comments? Visit the course discussion forum to chat with other learners.