Lab: Neural Networks for Music Classification

In addition to the concepts in the MNIST neural network demo, in this lab, you will learn to:

- Load a file from a URL
- Extract simple features from audio samples for machine learning tasks such as speech recognition and classification
- Build a simple neural network for music classification using these features
- Use a callback to store the loss and accuracy history in the training process
- Optimize the learning rate of the neural network

To illustrate the basic concepts, we will look at a relatively simple music classification problem. Given a sample of music, we want to determine which instrument (e.g. trumpet, violin, piano) is playing. This dataset was generously supplied by Prof. Juan Bello at NYU Stenihardt and his former PhD student Eric Humphrey (now at Spotify). They have a complete website dedicated to deep learning methods in music informatics:

http://marl.smusic.nyu.edu/wordpress/projects/feature-learning-deep-architectures/deep-learning-python-tutorial/

You can also check out Juan's course.

Loading Tensorflow

Before starting this lab, you will need to install Tensorflow. If you are using Google colaboratory, Tensorflow is already installed. Run the following command to ensure Tensorflow is installed.

In [ ]:

```
import tensorflow as tf
```

Then, load the other packages.

In [ ]:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
```

Audio Feature Extraction with Librosa

The key to audio classification is to extract the correct features. In addition to `keras`, we will need the `librosa` package. The `librosa` package in python has a rich set of methods extracting the features of audio samples commonly used in machine learning tasks such as speech recognition and sound classification.

Installation instructions and complete documentation for the package are given on the librosa main page. On most systems, you should be able to simply use:

```
pip install librosa
```

For Unix, you may need to load some additional packages:

```
sudo apt-get install build-essential
sudo apt-get install libxext-dev python-qt4 qt4-dev-tools
pip install librosa
```

After you have installed the package, try to import it.

In [ ]:
```
import librosa
import librosa.display
import librosa.feature
```

In this lab, we will use a set of music samples from the website:

http://theremin.music.uiowa.edu

This website has a great set of samples for audio processing. Look on the web for how to use the `requests.get` and `file.write` commands to load the file at the URL provided into your working directory.

You can play the audio sample by copying the file to your local machine and playing it on any media player. If you listen to it you will hear a soprano saxaphone (with vibrato) playing four notes (C, C#, D, Eb).

In [ ]:
```
import requests
fn = "SopSax.Vib.pp.C6Eb6.aiff"
url = "http://theremin.music.uiowa.edu/sound files/MIS/Woodwinds/sopranosaxophone/"+

# TODO:  Load the file from url and save it in a file under the name fn
request = requests.get(url)
with open(fn, 'wb') as f:
    f.write(request.content)
```

Next, use `librosa` command `librosa.load` to read the audio file with filename `fn` and get the samples `y` and sample rate `sr`.

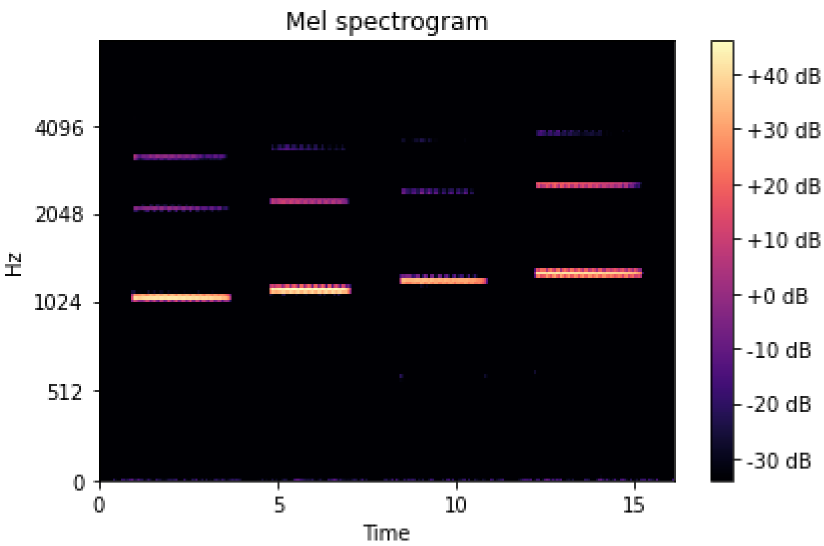
In [ ]:
```
# TODO
y, sr = librosa.load(fn)
```

Extracting features from audio files is an entire subject on its own right. A commonly used set of features are called the Mel Frequency Cepstral Coefficients (MFCCs). These are derived from the so-called mel spectrogram which is something like a regular spectrogram, but the power and frequency are represented in log scale, which more naturally aligns with human perceptual processing. You can run the code below to display the mel spectrogram from the audio sample.

You can easily see the four notes played in the audio track. You also see the 'harmonics' of each notes, which are other tones at integer multiples of the fundamental frequency of each note.

In [ ]:
```
S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128, fmax=8000)
librosa.display.specshow(librosa.amplitude_to_db(S),
                         y_axis='mel', fmax=8000, x_axis='time')
plt.colorbar(format='%+2.0f dB')
```

```
plt.title('Mel spectrogram')
plt.tight_layout()
```



Downloading the Data

Using the MFCC features described above, Eric Humphrey and Juan Bellow have created a complete data set that can used for instrument classification. Essentially, they collected a number of data files from the website above. For each audio file, the segmented the track into notes and then extracted 120 MFCCs for each note. The goal is to recognize the instrument from the 120 MFCCs. The process of feature extraction is quite involved. So, we will just use their processed data provided at:

https://github.com/marl/dl4mir-tutorial/blob/master/README.md

Note the password. Load the four files into some directory, say `instrument_dataset` . Then, load them with the commands.

In [ ]:

```
data_dir = './instrument_dataset/'
Xtr = np.load(data_dir+'uiowa_train_data.npy')
ytr = np.load(data_dir+'uiowa_train_labels.npy')
Xts = np.load(data_dir+'uiowa_test_data.npy')
yts = np.load(data_dir+'uiowa_test_labels.npy')
```

Looking at the data files:

- What are the number of training and test samples?
- What is the number of features for each sample?
- How many classes (i.e. instruments) are there per class?

In [ ]:

```
# TODO
print("Number of training samples: ", Xtr.shape[0])
print("Number of test samples: ", Xts.shape[0])
print("Number of features: ", Xtr.shape[1])
print("Number of classes: ", len(np.unique(ytr)))
```

```
Number of training samples:  66247
Number of test samples:  14904
```

```
Number of features:  120
Number of classes:  10
```

Before continuing, you must scale the training and test data, `Xtr` and `Xts` . Compute the mean and std deviation of each feature in `Xtr` and create a new training data set, `Xtr_scale` , by subtracting the mean and dividing by the std deviation. Also compute a scaled test data set, `Xts_scale` using the mean and std deviation learned from the training data set.

In [ ]:
```
# TODO Scale the training and test matrices
Xtr_scale = (Xtr- np.mean(Xtr)) / np.std(Xtr)
Xts_scale = (Xts- np.mean(Xtr)) / np.std(Xtr)
```

Building a Neural Network Classifier

Following the example in MNIST neural network demo, clear the keras session. Then, create a neural network `model` with:

- `nh=256` hidden units
- `sigmoid` activation
- select the input and output shapes correctly
- print the model summary

In [ ]:
```
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Activation
import tensorflow.keras.backend as K
```

In [ ]:
```
# TODO clear session
K.clear_session()
```

In [ ]:
```
# TODO: construct the model
nh = 256
nin = 120
nout = 10
model = Sequential()
model.add(Dense(units=nh, input_shape=(nin,), activation='sigmoid', name='hidden'))
model.add(Dense(units=nout, activation='sigmoid', name='output'))
```

In [ ]:
```
# TODO:  Print the model summary
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden (Dense)              (None, 256)               30976

 output (Dense)              (None, 10)                2570

=================================================================
Total params: 33,546
Trainable params: 33,546
Non-trainable params: 0
_________________________________________________________________
```

Create an optimizer and compile the model. Select the appropriate loss function and metrics. For the optimizer, use the Adam optimizer with a learning rate of 0.001

In [ ]:
```
# TODO
from tensorflow.keras import optimizers
opt = optimizers.Adam(lr=0.001)
model.compile(optimizer=opt, loss="sparse_categorical_crossentropy", metrics = ["acc
```

Fit the model for 10 epochs using the scaled data for both the training and validation. Use the `validation_data` option to pass the test data. Also, pass the callback class create above. Use a batch size of 100. Your final accuracy should be >99%.

In [ ]:
```
# TODO
hist = model.fit(Xtr_scale, ytr, epochs =10, batch_size =100, validation_data = (Xts
```

```
Epoch 1/10
663/663 [==============================] - 1s 856us/step - loss: 1.1890 - accuracy:
0.5963 - val_loss: 1.2155 - val_accuracy: 0.5925
Epoch 2/10
663/663 [==============================] - 1s 769us/step - loss: 0.7766 - accuracy:
0.7447 - val_loss: 0.8701 - val_accuracy: 0.7368
Epoch 3/10
663/663 [==============================] - 1s 785us/step - loss: 0.6029 - accuracy:
0.8008 - val_loss: 0.6939 - val_accuracy: 0.7815
Epoch 4/10
663/663 [==============================] - 1s 782us/step - loss: 0.4966 - accuracy:
0.8382 - val_loss: 0.5629 - val_accuracy: 0.8215
Epoch 5/10
663/663 [==============================] - 1s 783us/step - loss: 0.4205 - accuracy:
0.8668 - val_loss: 0.4925 - val_accuracy: 0.8546
Epoch 6/10
663/663 [==============================] - 1s 778us/step - loss: 0.3642 - accuracy:
0.8887 - val_loss: 0.4452 - val_accuracy: 0.8645
Epoch 7/10
663/663 [==============================] - 1s 782us/step - loss: 0.3203 - accuracy:
0.9038 - val_loss: 0.3794 - val_accuracy: 0.8824
Epoch 8/10
663/663 [==============================] - 1s 778us/step - loss: 0.2855 - accuracy:
0.9154 - val_loss: 0.3465 - val_accuracy: 0.8976
Epoch 9/10
663/663 [==============================] - 1s 797us/step - loss: 0.2580 - accuracy:
0.9241 - val_loss: 0.3266 - val_accuracy: 0.9090
Epoch 10/10
663/663 [==============================] - 1s 773us/step - loss: 0.2352 - accuracy:
0.9313 - val_loss: 0.3270 - val_accuracy: 0.9088
```

Plot the validation accuracy saved in `hist.history` dictionary. This gives one accuracy value per epoch. You should see that the validation accuracy saturates at a little higher than 99%. After that it "bounces around" due to the noise in the stochastic gradient descent.

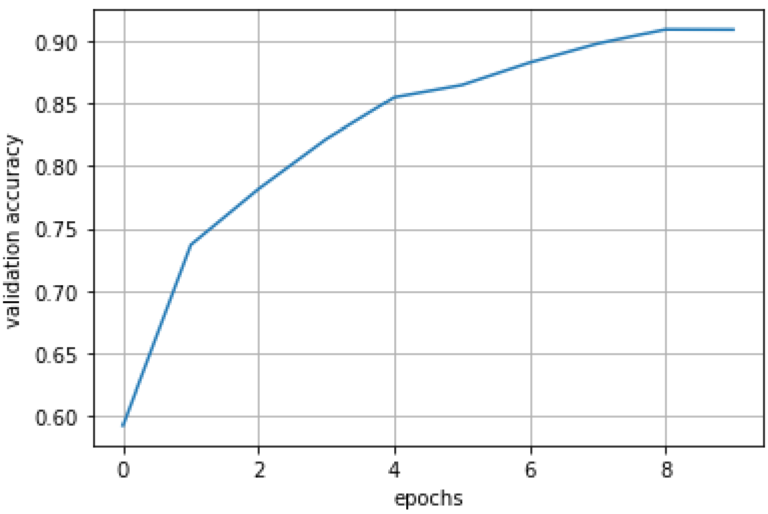
In [ ]:
```
hist.history
```

Out[ ]:
```
{'loss': [1.1890029907226562,
  0.7765828371047974,
  0.6028932332992554,
  0.4966089725494385,
  0.420482128858566З,
  0.3641979396343231,
  0.3203130960464775,
```

```
  0.2855477035045624,
  0.257972300052642 8,
  0.23517079651355743],
 'accuracy': [0.5963289141654968,
  0.7446677088737488,
  0.8008211851119995,
  0.8382115364074707,
  0.8668015003204346,
  0.8887044191360474,
  0.9038144946098328,
  0.9153621792793274,
  0.9240720272064209,
  0.9313478469848633],
 'val_loss': [1.2154797315597534,
  0.870056688785553,
  0.693882405757904,
  0.562886118888855,
  0.4924878180027008,
  0.44516047835350037,
  0.3794034421443939,
  0.3465077877044678,
  0.32656237483024597,
  0.3269965350627899],
 'val_accuracy': [0.5924584269523621,
  0.7367820739746094,
  0.7815351486206055,
  0.8214573264122009,
  0.8546028137207031,
  0.8644658923149109,
  0.8824476599693298,
  0.8976113796234131,
  0.9089506268501282,
  0.908816397190094]}
```

In [ ]:

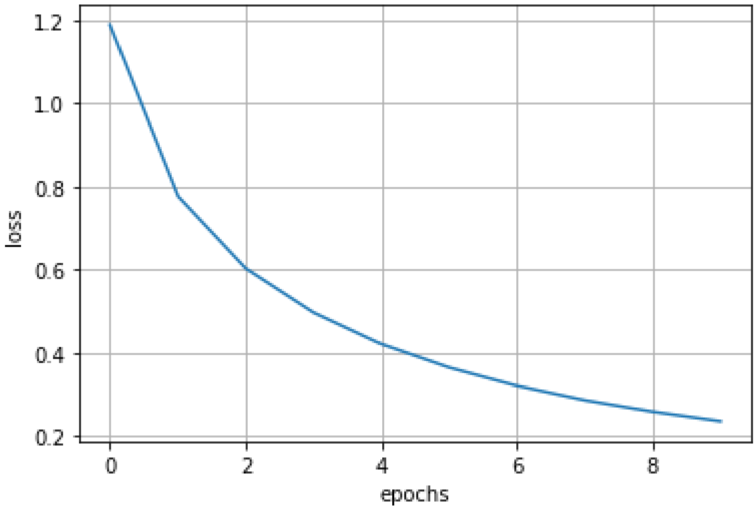
```
# TODO
accuracy_tr = hist.history['accuracy']
accuracy_ts = hist.history['val_accuracy']
plt.plot(accuracy_ts)
plt.xlabel("epochs")
plt.ylabel("validation accuracy")
plt.grid()
```



Plot the loss values saved in the `hist.history` dictionary. You should see that the loss is steadily decreasing. Use the `semilogy` plot.

In [ ]:

```
# TODO
loss = hist.history['loss']
plt.plot(loss)
plt.xlabel('epochs')
plt.ylabel('loss')
plt.grid()
```



Optimizing the Learning Rate

One challenge in training neural networks is the selection of the learning rate. Rerun the above code, trying four learning rates as shown in the vector `rates`. For each learning rate:

- clear the session
- construct the network
- select the optimizer. Use the Adam optimizer with the appropriate learrning rate.
- train the model for 20 epochs
- save the accuracy and losses

In [ ]:

```
rates = [0.01,0.001,0.0001]
batch_size = 100
loss_hist = []

# TODO
for lr in rates:
    K.clear_session()
    nh = 256
    nin = 120
    nout = 10
    model = Sequential()
    model.add(Dense(units=nh, input_shape=(nin,),activation='sigmoid', name='hidden')
    model.add(Dense(units=nout,activation='sigmoid', name='output'))
    opt = optimizers.Adam(lr)
    model.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metrics=['ac
    hist = model.fit(Xtr_scale, ytr, epochs =10, batch_size =100, validation_data = (
    loss_hist.append(hist.history['loss'])
```

```
Epoch 1/10
663/663 [==============================] - 1s 1ms/step - loss: 0.6498 - accuracy: 0.
7794 - val_loss: 0.4608 - val_accuracy: 0.8589
Epoch 2/10
663/663 [==============================] - 1s 853us/step - loss: 0.2642 - accuracy:
0.9120 - val_loss: 0.3060 - val_accuracy: 0.8983
```

```
Epoch 3/10
663/663 [==============================] - 1s 861us/step - loss: 0.1912 - accuracy: 0.9352 - val_loss: 0.2635 - val_accuracy: 0.9247
Epoch 4/10
663/663 [==============================] - 1s 884us/step - loss: 0.1508 - accuracy: 0.9496 - val_loss: 0.2903 - val_accuracy: 0.9261
Epoch 5/10
663/663 [==============================] - 1s 903us/step - loss: 0.1329 - accuracy: 0.9546 - val_loss: 0.3400 - val_accuracy: 0.9075
Epoch 6/10
663/663 [==============================] - 1s 906us/step - loss: 0.1186 - accuracy: 0.9599 - val_loss: 0.2657 - val_accuracy: 0.9259
Epoch 7/10
663/663 [==============================] - 1s 868us/step - loss: 0.1041 - accuracy: 0.9653 - val_loss: 0.2895 - val_accuracy: 0.9265
Epoch 8/10
663/663 [==============================] - 1s 866us/step - loss: 0.0978 - accuracy: 0.9675 - val_loss: 0.3533 - val_accuracy: 0.9216
Epoch 9/10
663/663 [==============================] - 1s 892us/step - loss: 0.0888 - accuracy: 0.9693 - val_loss: 0.4044 - val_accuracy: 0.8962
Epoch 10/10
663/663 [==============================] - 1s 873us/step - loss: 0.0864 - accuracy: 0.9708 - val_loss: 0.3518 - val_accuracy: 0.9280
Epoch 1/10
663/663 [==============================] - 1s 918us/step - loss: 1.1882 - accuracy: 0.5979 - val_loss: 1.2052 - val_accuracy: 0.5881
Epoch 2/10
663/663 [==============================] - 1s 830us/step - loss: 0.7719 - accuracy: 0.7461 - val_loss: 0.8688 - val_accuracy: 0.7250
Epoch 3/10
663/663 [==============================] - 1s 830us/step - loss: 0.5990 - accuracy: 0.8045 - val_loss: 0.6751 - val_accuracy: 0.7858
Epoch 4/10
663/663 [==============================] - 1s 821us/step - loss: 0.4939 - accuracy: 0.8400 - val_loss: 0.5562 - val_accuracy: 0.8378
Epoch 5/10
663/663 [==============================] - 1s 836us/step - loss: 0.4185 - accuracy: 0.8671 - val_loss: 0.4773 - val_accuracy: 0.8508
Epoch 6/10
663/663 [==============================] - 1s 837us/step - loss: 0.3618 - accuracy: 0.8902 - val_loss: 0.4219 - val_accuracy: 0.8784
Epoch 7/10
663/663 [==============================] - 1s 852us/step - loss: 0.3185 - accuracy: 0.9043 - val_loss: 0.3774 - val_accuracy: 0.8888
Epoch 8/10
663/663 [==============================] - 1s 835us/step - loss: 0.2834 - accuracy: 0.9179 - val_loss: 0.3581 - val_accuracy: 0.8970
Epoch 9/10
663/663 [==============================] - 1s 819us/step - loss: 0.2561 - accuracy: 0.9249 - val_loss: 0.3275 - val_accuracy: 0.9050
Epoch 10/10
663/663 [==============================] - 1s 829us/step - loss: 0.2335 - accuracy: 0.9334 - val_loss: 0.3093 - val_accuracy: 0.9085
Epoch 1/10
663/663 [==============================] - 1s 894us/step - loss: 1.7358 - accuracy: 0.4119 - val_loss: 1.8033 - val_accuracy: 0.3711
Epoch 2/10
663/663 [==============================] - 1s 841us/step - loss: 1.4359 - accuracy: 0.5058 - val_loss: 1.6915 - val_accuracy: 0.4135
Epoch 3/10
663/663 [==============================] - 1s 822us/step - loss: 1.3095 - accuracy: 0.5521 - val_loss: 1.6060 - val_accuracy: 0.4397
Epoch 4/10
```

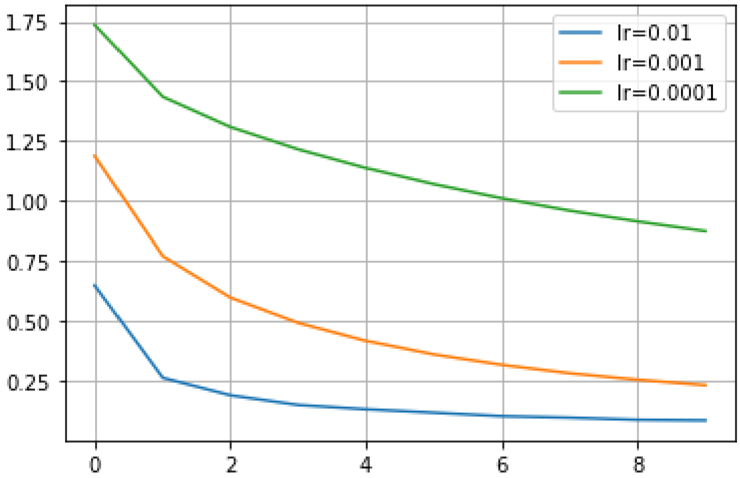
```
663/663 [==============================] - 1s 807us/step - loss: 1.2165 - accuracy: 0.5859 - val_loss: 1.5255 - val_accuracy: 0.4716
Epoch 5/10
663/663 [==============================] - 1s 804us/step - loss: 1.1384 - accuracy: 0.6146 - val_loss: 1.4450 - val_accuracy: 0.5020
Epoch 6/10
663/663 [==============================] - 1s 816us/step - loss: 1.0708 - accuracy: 0.6412 - val_loss: 1.3710 - val_accuracy: 0.5372
Epoch 7/10
663/663 [==============================] - 1s 828us/step - loss: 1.0121 - accuracy: 0.6638 - val_loss: 1.3141 - val_accuracy: 0.5448
Epoch 8/10
663/663 [==============================] - 1s 829us/step - loss: 0.9607 - accuracy: 0.6820 - val_loss: 1.2391 - val_accuracy: 0.5802
Epoch 9/10
663/663 [==============================] - 1s 855us/step - loss: 0.9156 - accuracy: 0.6983 - val_loss: 1.1899 - val_accuracy: 0.6094
Epoch 10/10
663/663 [==============================] - 1s 866us/step - loss: 0.8751 - accuracy: 0.7123 - val_loss: 1.1337 - val_accuracy: 0.6227
```

Plot the loss funciton vs. the epoch number for all three learning rates on one graph. You should see that the lower learning rates are more stable, but converge slower.

In [ ]:

```
# TODO
r1 = loss_hist[0]
r2 = loss_hist[1]
r3 = loss_hist[2]

plt.plot(r1)
plt.plot(r2)
plt.plot(r3)
plt.legend(['lr=0.01', 'lr=0.001', 'lr=0.0001'])
plt.grid()
```



In [ ]:

In [ ]: