Lab 7: Neural Networks for Music Classification

In addition to the concepts in the MNIST neural network demo (./mnist_neural.ipynb), 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 (http://steinhardt.nyu.edu/faculty/Juan_Pablo_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/ (http://marl.smusic.nyu.edu/wordpress/projects/feature-learning-deep-architectures/deep-learning-python-tutorial/)

You can also check out Juan's course (http://www.nyu.edu/classes/bello/ACA.html).

Loading the Keras package

We begin by loading keras and the other packages

In [1]:
```
import keras
```

```
Using TensorFlow backend.
```

In [2]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

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 (https://librosa.github.io/librosa/). On most systems, you should be able to simply use:

```
pip install -u 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 [3]:
```
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 (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 [4]:
```
import requests

fn = "SopSax.Vib.pp.C6Eb6.aiff"
url = "http://theremin.music.uiowa.edu/sound files/MIS/Woodwinds/sopranosaxo

# TODO:  Load the file from url and save it in a file under the name fn
r = requests.get(url)

import shutil

r = requests.get(url, stream=True)
if r.status_code == 200:
    with open(fn, 'wb') as f:
        r.raw.decode_content = True
        shutil.copyfileobj(r.raw, f)
```

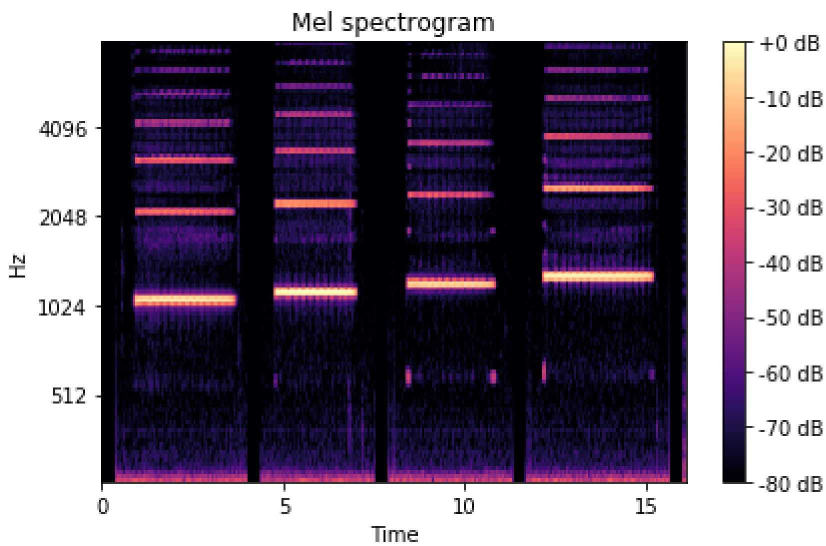
Next, use `librosa` command `librosa.load` to read the audio file with filename `fn` and get the samples `y` and sample rate `sr`.

In [5]:
```
# TODO
# y, sr = ...
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 [6]:
```
S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128, fmax=8000)
librosa.display.specshow(librosa.logamplitude(S,ref_power=np.max),
                         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 (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 [7]:
```
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 [8]:
```
# TODO
m_tr, nin, m_ts, _, nout = *Xtr.shape, *Xts.shape, len(np.unique(ytr))

print(("The training set has {} samples, while the "+
        "test set has {}.\nEach sample has {} features."+
        "There are {} classes (instruments).").format(m_tr, m_ts, nin, nout)
```

```
The training set has 66247 samples, while the test set has 14904.
Each sample has 120 features.There are 10 classes (instruments).
```

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 [9]:
```
# TODO Scale the training and test matrices
# Xtr_scale = ...
# Xts_scale = ...

mu = np.mean(Xtr, axis=0)
sigma = np.std(Xtr, axis=0)

Xtr_scale = (Xtr - mu) / sigma
Xts_scale = (Xts - mu) / sigma
```

Building a Neural Network Classifier

Following the example in MNIST neural network demo (./mnist_neural.ipynb), 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 [10]:
```
from keras.models import Model, Sequential
from keras.layers import Dense, Activation
```

In [11]:
```
# TODO clear session

import keras.backend as K
K.clear_session()
```

In [12]:
```
# TODO: construct the model
nh = 256

model = Sequential()
model.add(Dense(nh, input_shape=(nin,), activation='sigmoid', name='hidden')
model.add(Dense(nout, activation='softmax', name='output'))
```

In [13]:
```
# TODO:  Print the model summary
model.summary()
```

```
_________________________________________________________________
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
_________________________________________________________________
```

To keep track of the loss history and validation accuracy, we will use a *callback* function as described in [Keras callback documentation (https://keras.io/callbacks/)](https://keras.io/callbacks/). A callback is a class that is passed to the `fit` method. Complete the `LoadHistory` callback class below to save the loss and validation accuracy.

In [14]:
```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        # TODO:  Create two empty lists, self.loss and self.val_acc
        self.loss = []
        self.val_acc = []

    def on_batch_end(self, batch, logs={}):
        # TODO:  This is called at the end of each batch.
        # Add the loss in logs.get('loss') to the loss list
        self.loss.append( logs.get('loss') )

    def on_epoch_end(self, epoch, logs):
        # TODO:  This is called at the end of each epoch.
        # Add the test accuracy in logs.get('val_acc') to the val_acc list
        self.val_acc.append( logs.get('val_acc') )

# Create an instance of the history callback
history_cb = LossHistory()
```

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 [15]:
```
# TODO
# opt = ...
# model.compile(...)

from keras import optimizers

opt = optimizers.Adam(lr=0.001)
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

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 [16]:
```
# TODO
batch_size = 100

model.fit(Xtr_scale, ytr, epochs=10, batch_size=batch_size, validation_data=
```

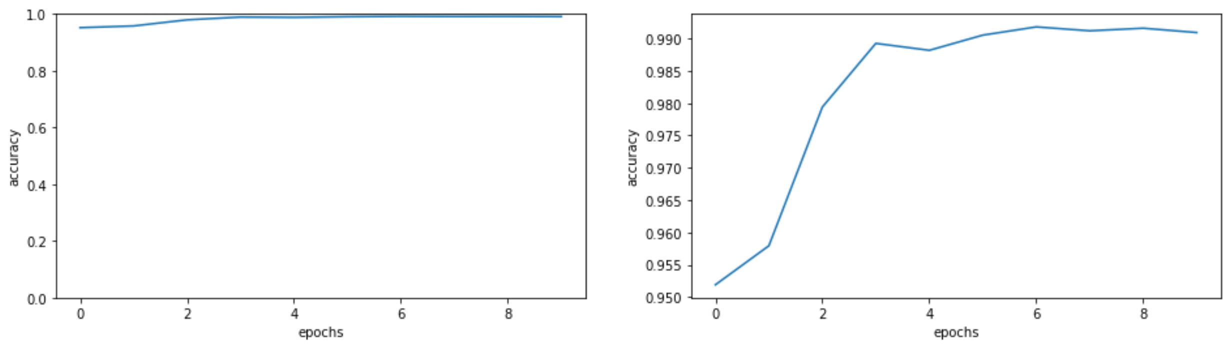
```
Train on 66247 samples, validate on 14904 samples
Epoch 1/10
66247/66247 [==============================] - 2s 30us/step - loss: 0.354
8 - acc: 0.9044 - val_loss: 0.1794 - val_acc: 0.9519
Epoch 2/10
66247/66247 [==============================] - 2s 27us/step - loss: 0.099
9 - acc: 0.9764 - val_loss: 0.1182 - val_acc: 0.9579
Epoch 3/10
66247/66247 [==============================] - 2s 29us/step - loss: 0.059
1 - acc: 0.9858 - val_loss: 0.0690 - val_acc: 0.9794
Epoch 4/10
66247/66247 [==============================] - 2s 30us/step - loss: 0.041
5 - acc: 0.9897 - val_loss: 0.0464 - val_acc: 0.9893
Epoch 5/10
66247/66247 [==============================] - 2s 31us/step - loss: 0.031
5 - acc: 0.9918 - val_loss: 0.0418 - val_acc: 0.9882
Epoch 6/10
66247/66247 [==============================] - 2s 28us/step - loss: 0.024
9 - acc: 0.9937 - val_loss: 0.0318 - val_acc: 0.9905
Epoch 7/10
66247/66247 [==============================] - 2s 32us/step - loss: 0.020
6 - acc: 0.9946 - val_loss: 0.0290 - val_acc: 0.9918
Epoch 8/10
66247/66247 [==============================] - 2s 31us/step - loss: 0.017
2 - acc: 0.9955 - val_loss: 0.0303 - val_acc: 0.9912
Epoch 9/10
66247/66247 [==============================] - 2s 30us/step - loss: 0.014
9 - acc: 0.9962 - val_loss: 0.0251 - val_acc: 0.9916
Epoch 10/10
66247/66247 [==============================] - 2s 30us/step - loss: 0.012
5 - acc: 0.9970 - val_loss: 0.0265 - val_acc: 0.9909
```

Out[16]:
```
<keras.callbacks.History at 0x11ae20a90>
```

Plot the validation accuracy saved in the `history_cb`. 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 [35]:
```
# TODO
plt.figure(figsize=(16,4))
plt.subplot(1,2,1)
plt.plot(history_cb.val_acc)
plt.ylabel("accuracy")
plt.xlabel("epochs")

axes = plt.gca()
axes.set_ylim([0,1])

plt.subplot(1,2,2)
plt.plot(history_cb.val_acc)
plt.ylabel("accuracy")
plt.xlabel("epochs")
```

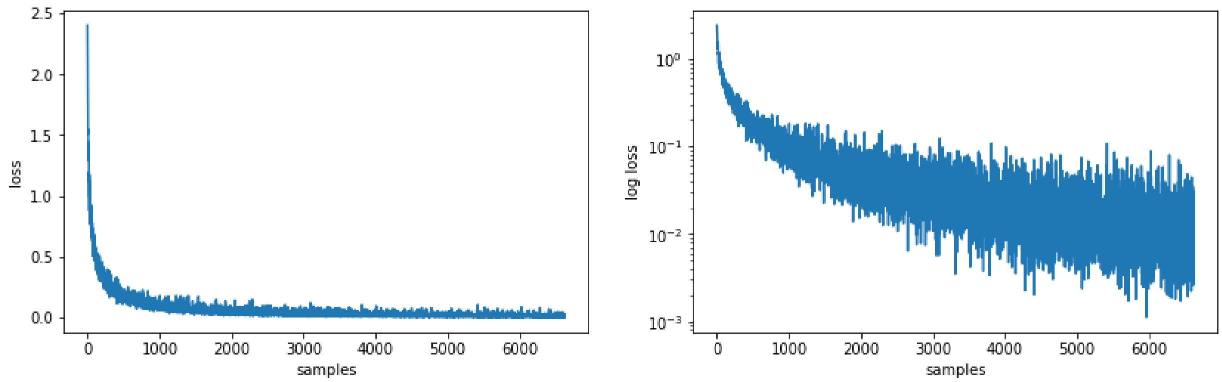
Out[35]: <matplotlib.text.Text at 0x12190c860>



Plot the loss values saved in the `history_cb` class. Use the `semilogy` plot. There is one loss value per step. But, plot the x-axis in epochs. Note that the epoch in step `i` is `epoch = i*batch_size/ntr` where `batch_size` is the batch_size and `ntr` is the total number of training samples.

In [38]:
```
# TODO
plt.figure(figsize=(14,4))

plt.subplot(1,2,1)
plt.plot(history_cb.loss)
plt.ylabel("loss")
plt.xlabel("samples")

plt.subplot(1,2,2)
plt.semilogy(history_cb.loss)
plt.ylabel("log loss")
plt.xlabel("samples")
```

Out[38]: <matplotlib.text.Text at 0x12236c4e0>



Optimizing the Learning Rate

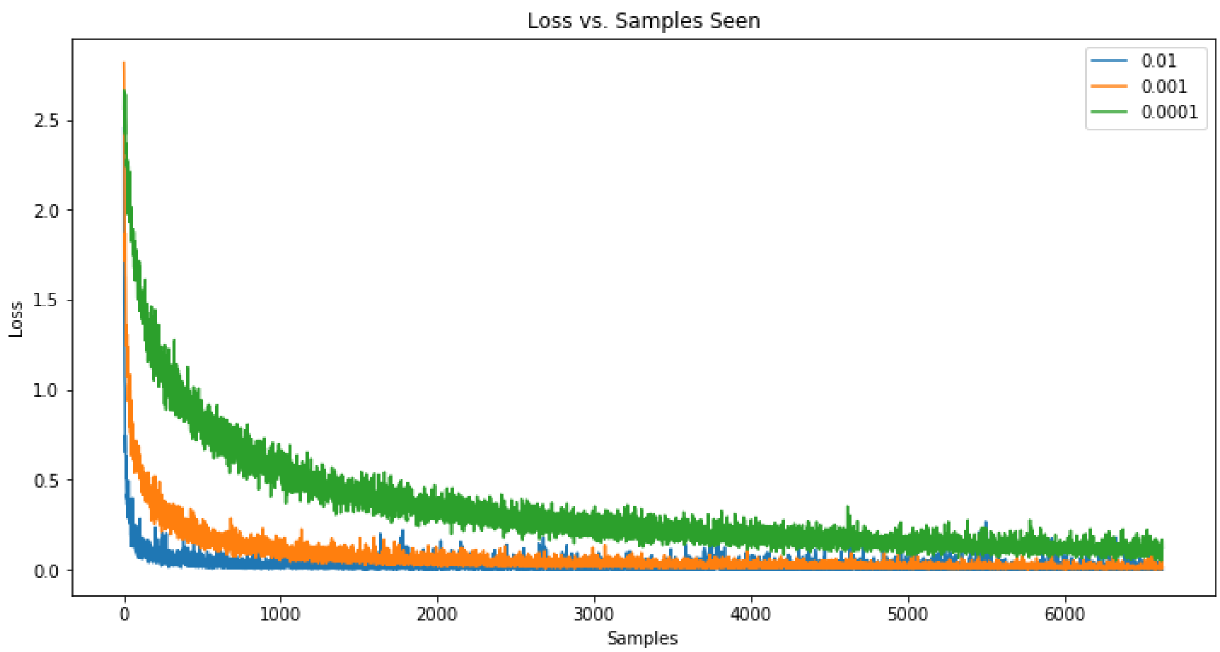
One challenge in training neural networks is the selection of the learning rate. Rerun the above code, trying three learning rates as shown in the vector `rates`. For each learning rate:

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

In [41]:
```
rates = [0.01,0.001,0.0001]
batch_size = 100
loss_hist = []
val_acc_hist = []

# TODO

for lr in rates:

    # clear session
    K.clear_session()

    # build network
    nh = 256

    model = Sequential()
    model.add(Dense(nh, input_shape=(nin,), activation='sigmoid', name='hidc
    model.add(Dense(nout, activation='softmax', name='output'))

    # define optimizer
    opt = optimizers.Adam(lr=lr)
    model.compile(optimizer=opt,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    # fit the model
    batch_size = 100
    history_cb = LossHistory()

    model.fit(Xtr_scale, ytr, epochs=10, batch_size=batch_size,
              validation_data=(Xts_scale,yts), callbacks=[history_cb], verbc

    loss_hist.append(history_cb.loss)
    val_acc_hist.append(history_cb.val_acc)
```

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 [44]:
```
# TODO
plt.figure(figsize=(12,6))
for i, lr in enumerate(rates):
    plt.title("Loss vs. Samples Seen")
    plt.plot(loss_hist[i], label=lr)
    plt.ylabel("Loss")
    plt.xlabel("Samples")
    plt.legend()
```



In [ ]:

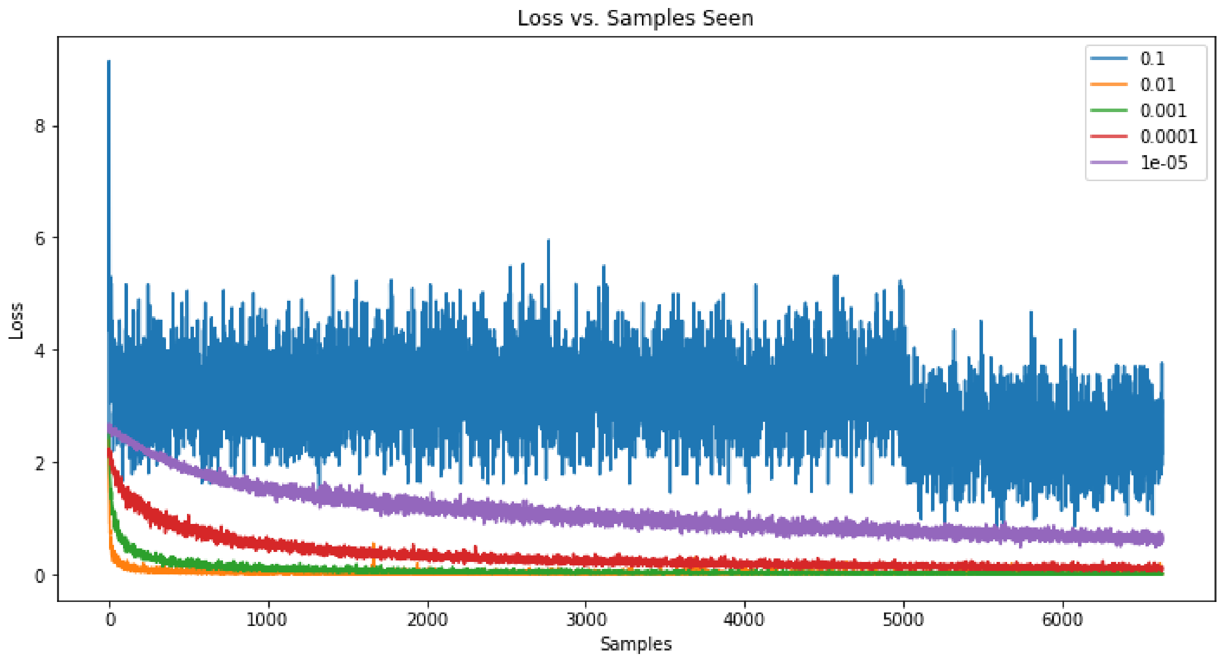
In [ ]:

In [47]:
```
rates = [0.1, 0.01,0.001,0.0001, 0.00001]
batch_size = 100
loss_hist = []
val_acc_hist = []

# TODO

for lr in rates:

    # clear session
    K.clear_session()

    # build network
    nh = 256

    model = Sequential()
    model.add(Dense(nh, input_shape=(nin,), activation='sigmoid', name='hidd
    model.add(Dense(nout, activation='softmax', name='output'))

    # define optimizer
    opt = optimizers.Adam(lr=lr)
    model.compile(optimizer=opt,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    # fit the model
    batch_size = 100
    history_cb = LossHistory()

    model.fit(Xtr_scale, ytr, epochs=10, batch_size=batch_size,
              validation_data=(Xts_scale,yts), callbacks=[history_cb], verbo

    loss_hist.append(history_cb.loss)
    val_acc_hist.append(history_cb.val_acc)
```

In [ ]:

In [48]:

```
# TODO
plt.figure(figsize=(12,6))
for i, lr in enumerate(rates):
    plt.title("Loss vs. Samples Seen")
    plt.plot(loss_hist[i], label=lr)
    plt.ylabel("Loss")
    plt.xlabel("Samples")
    plt.legend()
```



In [ ]: