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 [55]:
```
import keras
```

In [56]:
```
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 [57]:
```
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 [8]:
```
import requests
fn = "SopSax.Vib.pp.C6Eb6.aiff"
url = "http://theremin.music.uiowa.edu/sound files/MIS/Woodwinds/sopranosaxoph
one/"+fn

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

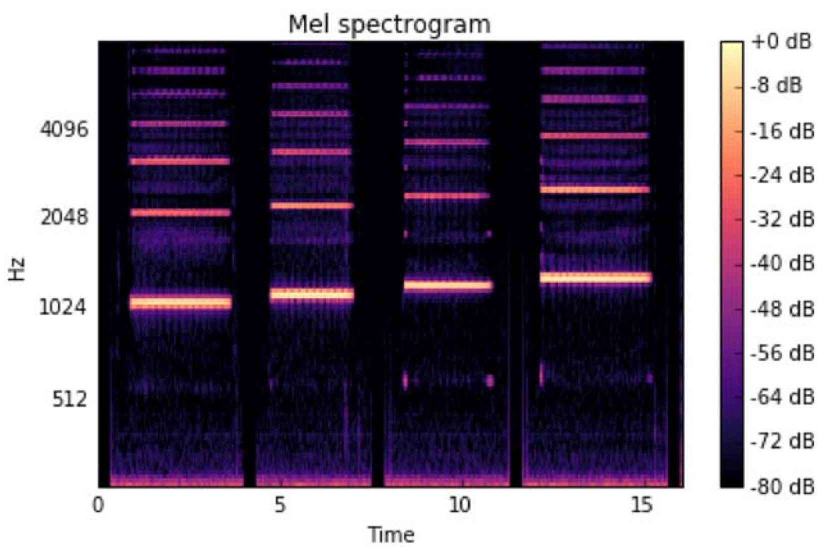
Next, use `librosa` command `librosa.load` to read the audio file with filename `fn` and get the samples `y` and sample rate `sr`.

In [9]:
```
# 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 [58]:
```
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)](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 [12]:
```
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 [59]:
```
# TODO
print('Num training= {0:d}'.format(Xtr.shape[0]))
print('Num test=     {0:d}'.format(Xts.shape[0]))
print('Num features= {0:d}'.format(Xtr.shape[1]))
print('Num classes=  {0:d}'.format(np.max(ytr)+1))
```

```
Num training= 66247
Num test=     14904
Num features= 120
Num 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 [67]:
```
# TODO Scale the training and test matrices
# Xtr_scale = ...
# Xts_scale = ...
xmean = np.mean(Xtr,axis=0)
xstd = np.std(Xtr,axis=0)
Xtr_scale = (Xtr-xmean[None,:])/xstd[None,:]
Xts_scale = (Xts-xmean[None,:])/xstd[None,:]
```

Building a Neural Network Classifier

Following the example in [MNIST neural network demo (./mnist_neural.ipynb)](./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 [75]:
```
from keras.models import Model, Sequential
from keras.layers import Dense, Activation
```

In [76]:
```
# TODO clear session
import keras.backend as K
K.clear_session()
```

In [77]:
```
# TODO: construct the model
nin = Xtr.shape[1]
nout = np.max(ytr)+1
nh = 256
model = Sequential()
model.add(Dense(nh, input_shape=(nin,), activation='sigmoid', name='hidden'))
model.add(Dense(nout, activation='softmax', name='output'))
```

In [78]:
```
# 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 [79]:
```
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('loss') 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 [80]:
```
# 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 [81]:
```
batch_size = 100
model.fit(Xtr_scale, ytr, epochs=10, batch_size=batch_size, validation_data=(X
ts_scale,yts), callbacks=[history_cb])
```

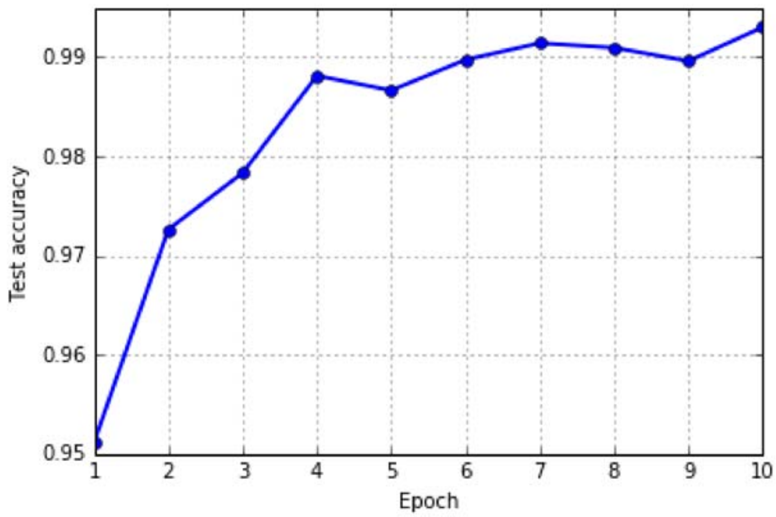
```
Train on 66247 samples, validate on 14904 samples
Epoch 1/10
66247/66247 [==============================] - 3s - loss: 0.3689 - acc: 0.899
1 - val_loss: 0.1872 - val_acc: 0.9512
Epoch 2/10
66247/66247 [==============================] - 2s - loss: 0.1058 - acc: 0.975
0 - val_loss: 0.0992 - val_acc: 0.9726
Epoch 3/10
66247/66247 [==============================] - 2s - loss: 0.0620 - acc: 0.985
2 - val_loss: 0.0710 - val_acc: 0.9783
Epoch 4/10
66247/66247 [==============================] - 2s - loss: 0.0438 - acc: 0.989
3 - val_loss: 0.0476 - val_acc: 0.9881
Epoch 5/10
66247/66247 [==============================] - 2s - loss: 0.0329 - acc: 0.991
6 - val_loss: 0.0461 - val_acc: 0.9866
Epoch 6/10
66247/66247 [==============================] - 2s - loss: 0.0261 - acc: 0.993
0 - val_loss: 0.0365 - val_acc: 0.9897
Epoch 7/10
66247/66247 [==============================] - 2s - loss: 0.0212 - acc: 0.994
4 - val_loss: 0.0292 - val_acc: 0.9914
Epoch 8/10
66247/66247 [==============================] - 2s - loss: 0.0177 - acc: 0.995
5 - val_loss: 0.0287 - val_acc: 0.9909
Epoch 9/10
66247/66247 [==============================] - 2s - loss: 0.0150 - acc: 0.996
2 - val_loss: 0.0328 - val_acc: 0.9896
Epoch 10/10
66247/66247 [==============================] - 2s - loss: 0.0130 - acc: 0.996
7 - val_loss: 0.0227 - val_acc: 0.9930
```

Out[81]: `<keras.callbacks.History at 0x1a23552aa20>`

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.

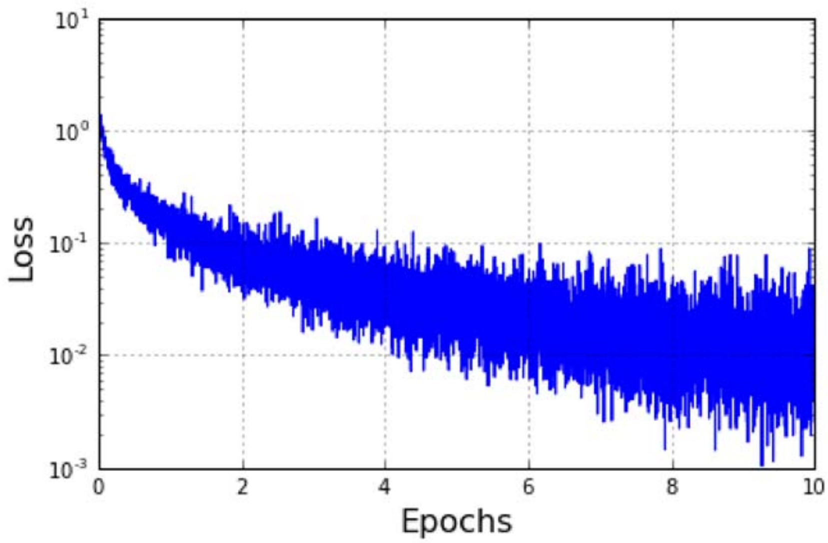
In [86]:
```
# TODO
val_acc = history_cb.val_acc
nepochs = len(val_acc)
plt.plot(np.arange(1,nepochs+1), val_acc, 'o-', linewidth=2)
plt.grid()
plt.xlabel('Epoch')
plt.ylabel('Test accuracy')
```

Out[86]: <matplotlib.text.Text at 0x1a234cab710>



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 [88]:
```
nsteps = len(history_cb.loss)
ntr = Xtr.shape[0]
epochs = np.arange(1,nsteps+1)*batch_size/ntr
plt.semilogy(epochs, history_cb.loss)
plt.xlabel('Epochs', fontsize=16)
plt.ylabel('Loss', fontsize=16)
plt.grid()
plt.xlim([0,np.max(epochs)])
plt.tight_layout()
```



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
- save the accuracy and losses

In [89]:
```
rates = [0.01,0.001,0.0001]
batch_size = 100
loss_hist = []
val_acc_hist = []

# TODO
for lr in rates:

    # Clear the session
    K.clear_session()

    # Build the model
    model = Sequential()
    model.add(Dense(nh, input_shape=(nin,), activation='sigmoid', name='hidde
n'))
    model.add(Dense(nout, activation='softmax', name='output'))

    # Select the optimizer with the correct learning rate to test
    opt = optimizers.Adam(lr=lr)
    model.compile(optimizer=opt,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

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

    # Get the losses
    loss_hist.append(history_cb.loss)
    val_acc_hist.append(history_cb.val_acc)

    # Print the final accuracy
    print("lr=%12.4e test accuracy=%f" % (lr, history_cb.val_acc[-1]))
```

```
Train on 66247 samples, validate on 14904 samples
Epoch 1/10
66247/66247 [==============================] - 2s - loss: 0.1111 - acc: 0.9659 - val_loss: 0.0674 - val_acc: 0.9754
Epoch 2/10
66247/66247 [==============================] - 2s - loss: 0.0279 - acc: 0.9910 - val_loss: 0.0396 - val_acc: 0.9862
Epoch 3/10
66247/66247 [==============================] - 2s - loss: 0.0218 - acc: 0.9929 - val_loss: 0.0383 - val_acc: 0.9871
Epoch 4/10
66247/66247 [==============================] - 2s - loss: 0.0195 - acc: 0.9936 - val_loss: 0.0560 - val_acc: 0.9802
Epoch 5/10
66247/66247 [==============================] - 2s - loss: 0.0172 - acc: 0.9942 - val_loss: 0.0244 - val_acc: 0.9928
Epoch 6/10
66247/66247 [==============================] - 2s - loss: 0.0133 - acc: 0.9958 - val_loss: 0.0223 - val_acc: 0.9919
Epoch 7/10
66247/66247 [==============================] - 2s - loss: 0.0131 - acc: 0.9958 - val_loss: 0.0280 - val_acc: 0.9906
Epoch 8/10
66247/66247 [==============================] - 2s - loss: 0.0148 - acc: 0.9950 - val_loss: 0.0679 - val_acc: 0.9826
Epoch 9/10
66247/66247 [==============================] - 2s - loss: 0.0111 - acc: 0.9964 - val_loss: 0.0606 - val_acc: 0.9817
Epoch 10/10
66247/66247 [==============================] - 2s - loss: 0.0105 - acc: 0.9969 - val_loss: 0.0558 - val_acc: 0.9844
lr=  1.0000e-02 test accuracy=0.984434
Train on 66247 samples, validate on 14904 samples
Epoch 1/10
66247/66247 [==============================] - 2s - loss: 0.3536 - acc: 0.9041 - val_loss: 0.1710 - val_acc: 0.9628
Epoch 2/10
66247/66247 [==============================] - 2s - loss: 0.1008 - acc: 0.9755 - val_loss: 0.0921 - val_acc: 0.9754
Epoch 3/10
66247/66247 [==============================] - 2s - loss: 0.0594 - acc: 0.9859 - val_loss: 0.0616 - val_acc: 0.9849
Epoch 4/10
66247/66247 [==============================] - 2s - loss: 0.0420 - acc: 0.9895 - val_loss: 0.0468 - val_acc: 0.9881
Epoch 5/10
66247/66247 [==============================] - 2s - loss: 0.0320 - acc: 0.9916 - val_loss: 0.0389 - val_acc: 0.9913
Epoch 6/10
66247/66247 [==============================] - 2s - loss: 0.0257 - acc: 0.9932 - val_loss: 0.0391 - val_acc: 0.9885
Epoch 7/10
66247/66247 [==============================] - 2s - loss: 0.0208 - acc: 0.9945 - val_loss: 0.0317 - val_acc: 0.9898
Epoch 8/10
66247/66247 [==============================] - 2s - loss: 0.0176 - acc: 0.9954 - val_loss: 0.0303 - val_acc: 0.9908
```

```
Epoch 9/10
66247/66247 [==============================] - 2s - loss: 0.0149 - acc:
0.9962 - val_loss: 0.0256 - val_acc: 0.9924
Epoch 10/10
66247/66247 [==============================] - 2s - loss: 0.0133 - acc:
0.9963 - val_loss: 0.0268 - val_acc: 0.9914
lr=  1.0000e-03 test accuracy=0.991412
Train on 66247 samples, validate on 14904 samples
Epoch 1/10
66247/66247 [==============================] - 2s - loss: 1.0863 - acc:
0.6680 - val_loss: 0.8283 - val_acc: 0.6824
Epoch 2/10
66247/66247 [==============================] - 2s - loss: 0.5394 - acc:
0.8517 - val_loss: 0.5572 - val_acc: 0.8292
Epoch 3/10
66247/66247 [==============================] - 2s - loss: 0.3743 - acc:
0.9141 - val_loss: 0.4214 - val_acc: 0.8865
Epoch 4/10
66247/66247 [==============================] - 3s - loss: 0.2895 - acc:
0.9352 - val_loss: 0.3405 - val_acc: 0.9081
Epoch 5/10
66247/66247 [==============================] - 3s - loss: 0.2356 - acc:
0.9474 - val_loss: 0.2799 - val_acc: 0.9255
Epoch 6/10
66247/66247 [==============================] - 2s - loss: 0.1969 - acc:
0.9556 - val_loss: 0.2361 - val_acc: 0.9351
Epoch 7/10
66247/66247 [==============================] - 2s - loss: 0.1674 - acc:
0.9611 - val_loss: 0.2047 - val_acc: 0.9416
Epoch 8/10
66247/66247 [==============================] - 2s - loss: 0.1442 - acc:
0.9660 - val_loss: 0.1730 - val_acc: 0.9529
Epoch 9/10
66247/66247 [==============================] - 2s - loss: 0.1257 - acc:
0.9698 - val_loss: 0.1527 - val_acc: 0.9571
Epoch 10/10
66247/66247 [==============================] - 2s - loss: 0.1107 - acc:
0.9736 - val_loss: 0.1342 - val_acc: 0.9629
lr=  1.0000e-04 test accuracy=0.962896
```

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 [91]:

```
# TODO
ntest = len(loss_hist)
ntr = Xtr.shape[0]
batch_size=100
for it, loss in enumerate(loss_hist):
    nsteps = len(loss)
    epochs = np.arange(nsteps)*batch_size/ntr

    plt.semilogy(epochs, loss)

    rate_str = ['{0:5.4f}'.format(lr) for lr in rates]

plt.axis([0,np.max(epochs),1e-5,10])
plt.xlabel('Epochs', fontsize=16)
plt.ylabel('Loss', fontsize=16)
plt.legend(rate_str,loc='lower left')
plt.tight_layout()
```