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
- Record 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/)

Installing and Loading the Keras package ¶

Before beginning you need to make sure you have installed tensorflow and keras. Full instructions are available:

- Tensorflow installation page (https://www.tensorflow.org/install/)
- Keras installation page (https://keras.io/#installation)

For the most part, you should be able to install both with the commands:

```
pip3 install --upgrade tensorflow
pip3 install --upgrade keras
```

After you have installed the packages, we can begin by loading keras and the other packages

In [6]:
```
import keras
```

In [5]:
```
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/)](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 [7]:
```
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)](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 [16]:
```
import requests
fn = "SopSax.Vib.pp.C6Eb6.aiff"
url = "http://theremin.music.uiowa.edu/sound files/MIS/Woodwinds/sopranosaxophone/"+fn

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

Out[16]:
```
<Response [200]>
```

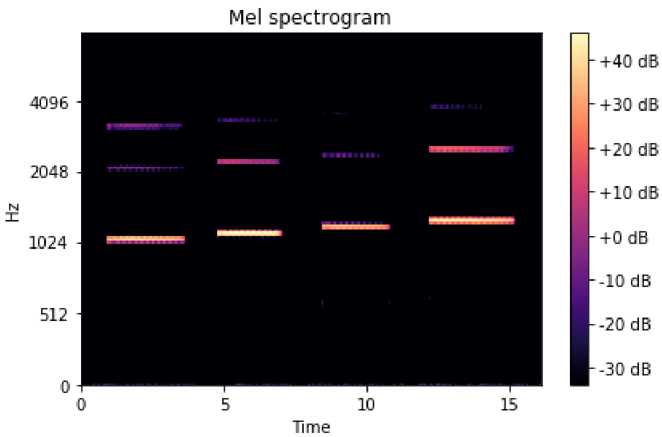
Next, use `librosa` command `librosa.load` to read the audio file with filename `fn` and get the samples `y` and sample rate `sr`.

In [17]:
```
# 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 [18]:
```
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 (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 [25]:
```
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 [138]:

```
# TODO
print(Xtr.shape)
print(Xts.shape)
print(ytr.shape)
print(yts.shape)
print(np.unique(ytr))
print(np.unique(yts))

# Thus, there are 66247 training samples and 14904 testing samples (about 20% of total samples)
# There are 120 features for each sample
# There are 10 classes total. The number of instruments per class over both training and test is given
 below.
print(sum(ytr == 0) + sum(yts == 0)) # 8206 in class 0
print(sum(ytr == 1) + sum(yts == 1)) # 2608 in class 1
print(sum(ytr == 2) + sum(yts == 2)) # 5038 in class 2
print(sum(ytr == 3) + sum(yts == 3)) # 10763 in class 3
print(sum(ytr == 4) + sum(yts == 4)) # 5117 in class 4
print(sum(ytr == 5) + sum(yts == 5)) # 11354 in class 5
print(sum(ytr == 6) + sum(yts == 6)) # 14699 in class 6
print(sum(ytr == 7) + sum(yts == 7)) # 3143 in class 7
print(sum(ytr == 8) + sum(yts == 8)) # 2613 in class 8
print(sum(ytr == 9) + sum(yts == 9)) # 17610 in class 9
```

```
(66247, 120)
(14904, 120)
(66247,)
(14904,)
[0 1 2 3 4 5 6 7 8 9]
[0 1 2 3 4 5 6 7 8 9]
8206
2608
5038
10763
5117
11354
14699
3143
2613
17610
```

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 [123]:

```
# TODO Scale the training and test matrices

# find means and stds of Xtr
col_mean = np.mean(Xtr, axis = 0)
col_std = np.std(Xtr, axis = 0)

# scale Xtr and Xts
Xtr_scale = (Xtr - col_mean) / col_std
Xts_scale = (Xts - col_mean) / col_std

# ensuring scaling worked
print(Xtr_scale.mean())
print(Xtr_scale.std())
print(Xts_scale.mean())
print(Xts_scale.std())
```

```
-1.8859998891429954e-16
0.9999999999999998
-0.060611038815661974
0.915019690809686
```

Building a Neural Network Classifier

Following the example in MNIST neural network demo posted on CCLE on May 24th (there is both a pdf file and a python notebook file; you can save the python notebook file and open it in Jupyter or similiar tool), 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 [124]:
```
from keras.models import Model, Sequential
from keras.layers import Dense, Activation
```

In [125]:
```
# Clear session (removes existing models)
import keras.backend as K  # From demo (so you can use K.xxx instead of typing keras.backend.xxx)
K.clear_session()
```

In [126]:
```
# TODO: construct the model
nin = Xtr.shape[1] # dimension of input data = 120 since there are 120 features
nh = 256 # number of hidden units
nout = 10 # number of outputs = 10 since there are 10 classes
model = Sequential()
model.add(Dense(nh, input_shape = (nin, ), activation = "sigmoid", name = "hidden"))
model.add(Dense(nout, activation = "softmax", name = "output"))
```

In [127]:
```
# 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
_________________________________________________________________
```

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 [128]:
```
# TODO
from keras import optimizers

opt = optimizers.Adam(lr = 0.001) # beta_1 = 0.9, beta_2 = 0.999, epsilon = 1e-08, decay = 0.0
model.compile(optimizer = opt,
              loss = "sparse_categorical_crossentropy",
              metrics = ["accuracy"]) # since this is a multi-class classification problem
```

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. Use a batch size of 100. Your final accuracy should be >99%. To record the training history, use

```
hist = model.fit(...)
```

This will return a data structure, `hist` with the metrics per epoch.

```
In [129]: # TODO
hist = model.fit(Xtr_scale, ytr, epochs = 10, batch_size = 100, validation_data = (Xts_scale, yts))
hist # final accuracys is above 99% - yay!
```

```
Train on 66247 samples, validate on 14904 samples
Epoch 1/10
66247/66247 [==============================] - 3s 44us/step - loss: 0.3612 - acc: 0.9016 - val_loss:
0.1892 - val_acc: 0.9494
Epoch 2/10
66247/66247 [==============================] - 3s 46us/step - loss: 0.1047 - acc: 0.9744 - val_loss:
0.0959 - val_acc: 0.9738
Epoch 3/10
66247/66247 [==============================] - 3s 43us/step - loss: 0.0617 - acc: 0.9856 - val_loss:
0.0748 - val_acc: 0.9786
Epoch 4/10
66247/66247 [==============================] - 3s 43us/step - loss: 0.0427 - acc: 0.9893 - val_loss:
0.0523 - val_acc: 0.9877
Epoch 5/10
66247/66247 [==============================] - 3s 47us/step - loss: 0.0323 - acc: 0.9922 - val_loss:
0.0422 - val_acc: 0.9882
Epoch 6/10
66247/66247 [==============================] - 3s 42us/step - loss: 0.0252 - acc: 0.9935 - val_loss:
0.0328 - val_acc: 0.9905
Epoch 7/10
66247/66247 [==============================] - 3s 43us/step - loss: 0.0207 - acc: 0.9947 - val_loss:
0.0325 - val_acc: 0.9899
Epoch 8/10
66247/66247 [==============================] - 3s 43us/step - loss: 0.0174 - acc: 0.9955 - val_loss:
0.0327 - val_acc: 0.9896
Epoch 9/10
66247/66247 [==============================] - 3s 42us/step - loss: 0.0146 - acc: 0.9963 - val_loss:
0.0259 - val_acc: 0.9917
Epoch 10/10
66247/66247 [==============================] - 3s 45us/step - loss: 0.0129 - acc: 0.9969 - val_loss:
0.0276 - val_acc: 0.9899
```

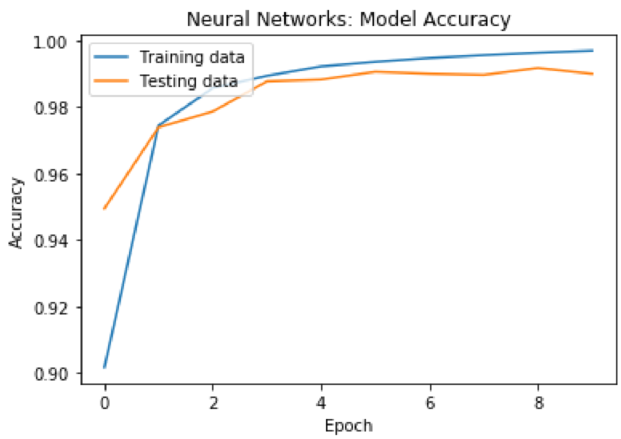
```
Out[129]: <keras.callbacks.History at 0x2371024af28>
```

The data structure `hist` should contain the training and validation accuracy as a function of the epoch:

- `hist.history['acc']`: Training accuracy per epoch
- `hist.history['val_acc']`: Test/validation accuracy per epoch

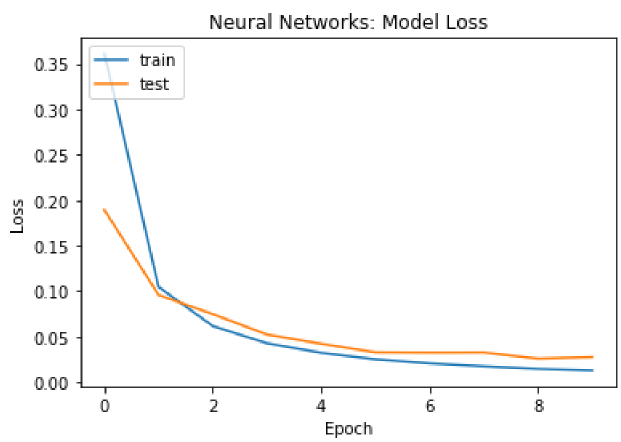
Plot the training and test validation as a function of the epoch. Label your graph.

In [130]:
```
# TODO
plt.plot(hist.history['acc'])
plt.plot(hist.history['val_acc'])
plt.title("Neural Networks: Model Accuracy")
plt.ylabel("Accuracy")
plt.xlabel("Epoch")
plt.legend(["Training data", "Testing data"], loc = "upper left")
plt.show()
```



Plot the loss function, stored in `hist.history['loss']`.

In [131]:
```
# TODO
plt.plot(hist.history["loss"])
plt.plot(hist.history["val_loss"])
plt.title("Neural Networks: Model Loss")
plt.ylabel("Loss")
plt.xlabel("Epoch")
plt.legend(['train', 'test'], loc = "upper left")
plt.show()
```



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 [132]:
```
rates = [0.01,0.001,0.0001]
batch_size = 100
loss_hist = []
val_acc_hist = []

# TODO

for i in range(0, 3):
    # Clear session (removes existing models)
    import keras.backend as K  # From demo (so you can use K.xxx instead of typing keras.backend.xx
x)
    K.clear_session()

    # construct the model
    nin = Xtr.shape[1] # dimension of input data = 120 since there are 120 features
    nh = 256 # number of hidden units
    nout = 10 # number of outputs = 10 since there are 10 classes
    model = Sequential()
    model.add(Dense(nh, input_shape = (nin, ), activation = "sigmoid", name = "hidden"))
    model.add(Dense(nout, activation = "softmax", name = "output"))

    # select the optimizer
    from keras import optimizers

    opt = optimizers.Adam(lr = rates[i]) # beta_1 = 0.9, beta_2 = 0.999, epsilon = 1e-08, decay =
 0.0
    model.compile(optimizer = opt,
                  loss = "sparse_categorical_crossentropy",
                  metrics = ["accuracy"]) # since this is a multi-class classification problem

    # train the model
    hist = model.fit(Xtr_scale, ytr, epochs = 10, batch_size = 100, validation_data = (Xts_scale, y
ts))

    # save accuracy and loss functions
    loss_hist = np.append(loss_hist, hist.history["loss"])
    val_acc_hist = np.append(val_acc_hist, hist.history["val_acc"])

# view accuracy and lost arrays
print(loss_hist)
print(val_acc_hist)
```

```
Train on 66247 samples, validate on 14904 samples
Epoch 1/10
66247/66247 [==============================] - 3s 49us/step - loss: 0.1120 - acc: 0.9662 - val_los
s: 0.0465 - val_acc: 0.9843
Epoch 2/10
66247/66247 [==============================] - 4s 62us/step - loss: 0.0280 - acc: 0.9908 - val_los
s: 0.0305 - val_acc: 0.9894
Epoch 3/10
66247/66247 [==============================] - 4s 60us/step - loss: 0.0200 - acc: 0.9937 - val_los
s: 0.0310 - val_acc: 0.9888
Epoch 4/10
66247/66247 [==============================] - 4s 54us/step - loss: 0.0172 - acc: 0.9939 - val_los
s: 0.0325 - val_acc: 0.9881
Epoch 5/10
66247/66247 [==============================] - 3s 44us/step - loss: 0.0169 - acc: 0.9946 - val_los
s: 0.0504 - val_acc: 0.9858
Epoch 6/10
66247/66247 [==============================] - 3s 43us/step - loss: 0.0162 - acc: 0.9948 - val_los
s: 0.0468 - val_acc: 0.9852
Epoch 7/10
66247/66247 [==============================] - 3s 44us/step - loss: 0.0141 - acc: 0.9953 - val_los
s: 0.0489 - val_acc: 0.9832
Epoch 8/10
66247/66247 [==============================] - 3s 44us/step - loss: 0.0121 - acc: 0.9960 - val_los
s: 0.0501 - val_acc: 0.9862
Epoch 9/10
66247/66247 [==============================] - 3s 49us/step - loss: 0.0113 - acc: 0.9967 - val_los
s: 0.0777 - val_acc: 0.9806
Epoch 10/10
66247/66247 [==============================] - 3s 45us/step - loss: 0.0097 - acc: 0.9970 - val_los
s: 0.0439 - val_acc: 0.9871
Train on 66247 samples, validate on 14904 samples
Epoch 1/10
66247/66247 [==============================] - 4s 63us/step - loss: 0.3638 - acc: 0.9006 - val_los
s: 0.1843 - val_acc: 0.9486
Epoch 2/10
66247/66247 [==============================] - 4s 60us/step - loss: 0.1043 - acc: 0.9751 - val_los
s: 0.1206 - val_acc: 0.9578
Epoch 3/10
66247/66247 [==============================] - 4s 56us/step - loss: 0.0612 - acc: 0.9851 - val_los
s: 0.0653 - val_acc: 0.9838
Epoch 4/10
66247/66247 [==============================] - 4s 60us/step - loss: 0.0430 - acc: 0.9889 - val_los
s: 0.0501 - val_acc: 0.9866
Epoch 5/10
66247/66247 [==============================] - 4s 63us/step - loss: 0.0324 - acc: 0.9916 - val_los
s: 0.0386 - val_acc: 0.9899
Epoch 6/10
66247/66247 [==============================] - 3s 50us/step - loss: 0.0258 - acc: 0.9934 - val_los
s: 0.0492 - val_acc: 0.9840
Epoch 7/10
66247/66247 [==============================] - 3s 52us/step - loss: 0.0208 - acc: 0.9947 - val_los
s: 0.0313 - val_acc: 0.9913
Epoch 8/10
66247/66247 [==============================] - 3s 52us/step - loss: 0.0174 - acc: 0.9954 - val_los
s: 0.0308 - val_acc: 0.9901
Epoch 9/10
66247/66247 [==============================] - 3s 51us/step - loss: 0.0151 - acc: 0.9960 - val_los
s: 0.0243 - val_acc: 0.9926
Epoch 10/10
66247/66247 [==============================] - 3s 46us/step - loss: 0.0130 - acc: 0.9966 - val_los
s: 0.0258 - val_acc: 0.9919
Train on 66247 samples, validate on 14904 samples
Epoch 1/10
66247/66247 [==============================] - 3s 44us/step - loss: 1.0938 - acc: 0.6581 - val_los
s: 0.8256 - val_acc: 0.6893
Epoch 2/10
66247/66247 [==============================] - 3s 46us/step - loss: 0.5345 - acc: 0.8615 - val_los
s: 0.5576 - val_acc: 0.8352
Epoch 3/10
66247/66247 [==============================] - 3s 45us/step - loss: 0.3720 - acc: 0.9165 - val_los
```

```
s: 0.4285 - val_acc: 0.8785
Epoch 4/10
66247/66247 [==============================] - 3s 43us/step - loss: 0.2885 - acc: 0.9363 - val_los
s: 0.3472 - val_acc: 0.9035
Epoch 5/10
66247/66247 [==============================] - 3s 44us/step - loss: 0.2349 - acc: 0.9484 - val_los
s: 0.2821 - val_acc: 0.9238
Epoch 6/10
66247/66247 [==============================] - 3s 43us/step - loss: 0.1961 - acc: 0.9561 - val_los
s: 0.2301 - val_acc: 0.9414
Epoch 7/10
66247/66247 [==============================] - 3s 44us/step - loss: 0.1664 - acc: 0.9621 - val_los
s: 0.1988 - val_acc: 0.9485
Epoch 8/10
66247/66247 [==============================] - 3s 48us/step - loss: 0.1432 - acc: 0.9669 - val_los
s: 0.1718 - val_acc: 0.9554
Epoch 9/10
66247/66247 [==============================] - 3s 44us/step - loss: 0.1246 - acc: 0.9712 - val_los
s: 0.1509 - val_acc: 0.9588
Epoch 10/10
66247/66247 [==============================] - 3s 44us/step - loss: 0.1096 - acc: 0.9747 - val_los
s: 0.1384 - val_acc: 0.9589
[0.11197145 0.02797692 0.02001501 0.01724713 0.01690814 0.01616939
 0.01414337 0.01214547 0.01128055 0.00974505 0.36379008 0.10426827
 0.06115358 0.04301714 0.03238684 0.02584252 0.02078201 0.01742805
 0.01509408 0.0130124  1.09384679 0.53448608 0.37199049 0.28852188
 0.2348947  0.1960534  0.16640836 0.14319898 0.12457257 0.10955078]
[0.98429952 0.98939882 0.98879496 0.9880569  0.98584273 0.98517177
 0.98322598 0.98624531 0.98060923 0.98705046 0.9486044  0.95779657
 0.98376275 0.98664789 0.9898685  0.98396404 0.99134461 0.99013688
 0.99261943 0.99188138 0.68927805 0.83521202 0.878489   0.90351584
 0.92384595 0.94142512 0.94853731 0.95544821 0.95880301 0.9588701 ]
```

Plot the loss function vs. the epoch number for all three learning rates on one graph. In order that you can see the difference, it may be useful to plot them using `semilogy` so that the loss is in log scale. You should see that the lower learning rates are more stable, but converge slower.

In [134]:
```
# TODO
matplotlib.pyplot.semilogy(loss_hist[0:9])
matplotlib.pyplot.semilogy(loss_hist[10:19])
matplotlib.pyplot.semilogy(loss_hist[20:29])
plt.title("Neural Networks: Model Loss")
plt.ylabel("Loss")
plt.xlabel("Epoch")
plt.legend(["lr = 0.01", "lr = 0.001", "lr = 0.0001"], loc = "upper left")
plt.show()
```