lab_music_partial

November 15, 2021

1 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.

1.1 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.

conda activate tf2

[1]:
```
import tensorflow as tf
```

Then, load the other packages.

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

1.2 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.

[3]:
```python
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).

[4]:
```python
import requests
fn = "SopSax.Vib.pp.C6Eb6.aiff"
url = "http://theremin.music.uiowa.edu/sound files/MIS/Woodwinds/
 ↪sopranosaxophone/"+fn

# TODO 1:  Load the file from url and save it in a file under the name fn
```

[5]:
```python
r = requests.get(url)
with open(fn, 'wb') as f:
    f.write(r.content)
```

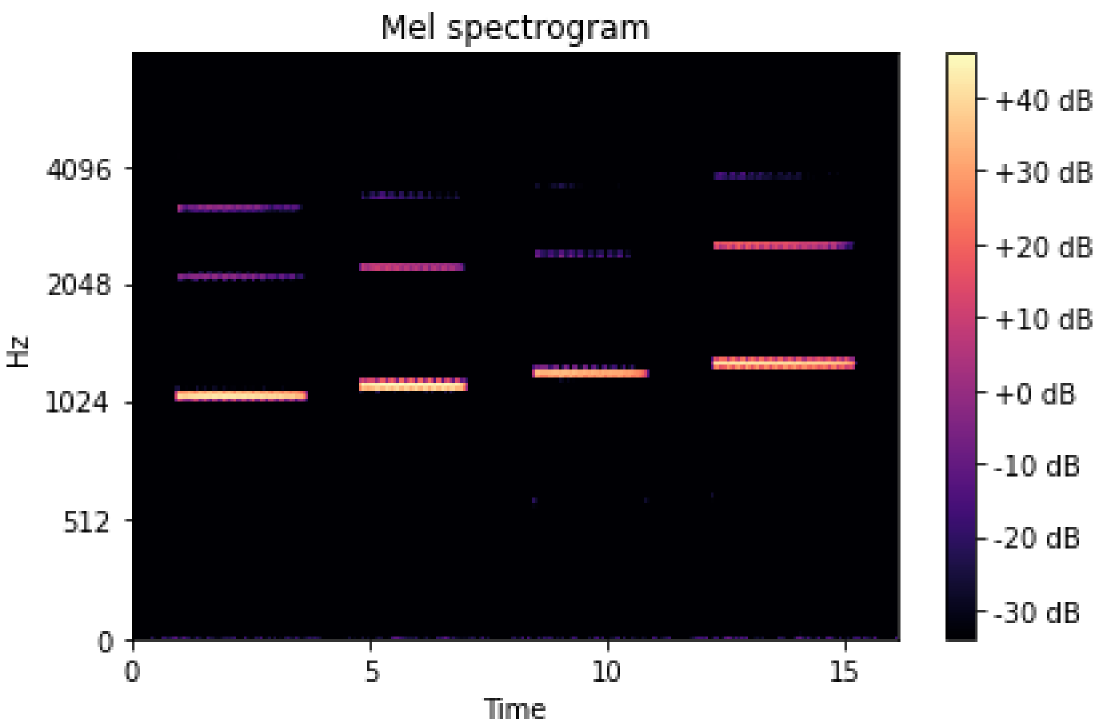
Next, use `librosa` command `librosa.load` to read the audio file with filename `fn` and get the samples `y` and sample rate `sr`.

[6]:
```python
# TODO 2
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.

[7]:
```
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()
```



1.3 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.

[8]:
```
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?

[9]:
```
# TODO 3
num_train_sample, num_train_feature = np.shape(Xtr)
num_test_sample, num_test_feature = np.shape(Xts)
num_classes = len(np.unique(ytr))

print('num train sample = '+str(num_train_sample))
print('num test sample = '+str(num_test_sample))
print('num feature for each sample = '+str(num_train_feature))
print('num classes = '+str(num_classes))
```

```
num train sample = 66247
num test sample = 14904
num feature for each sample = 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.

[10]:
```
# TODO 4: Scale the training and test matrices
# Xtr_scale = ...
# Xts_scale = ...
```

[11]:
```
Xtr_mean = np.mean(Xtr, axis = 0)
Xtr_std = np.std(Xtr, axis = 0)
```

[12]:
```
Xtr_scale = (Xtr - Xtr_mean)/Xtr_std
Xts_scale = (Xts - Xtr_mean)/Xtr_std
```

1.4 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

[13]:
```
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Activation
import tensorflow.keras.backend as K
```

[14]:
```
# TODO 5 clear session
K.clear_session()
```

[15]:
```
# TODO 6: construct the model
nin = num_train_feature  # dimension of input data
nh = 256      # number of hidden units
nout = num_classes    # number of outputs = 10 since there are 10 classes
model = Sequential()
model.add(Dense(units=nh, input_shape=(nin,), activation='sigmoid',␣
 ↪name='hidden'))
model.add(Dense(units=nout, activation='softmax', name='output'))
```

```
2021-11-15 09:46:03.666941: I tensorflow/core/platform/cpu_feature_guard.cc:145]
This TensorFlow binary is optimized with Intel(R) MKL-DNN to use the following
CPU instructions in performance critical operations:  SSE4.1 SSE4.2
To enable them in non-MKL-DNN operations, rebuild TensorFlow with the
appropriate compiler flags.
2021-11-15 09:46:03.667454: I
tensorflow/core/common_runtime/process_util.cc:115] Creating new thread pool
with default inter op setting: 8. Tune using inter_op_parallelism_threads for
best performance.
```

[16]:
```
# TODO 7:  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

[17]:
```
# TODO 8
# opt = ...
# model.compile(...)
```

[18]:
```
from tensorflow.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%.

```
[19]: # TODO 9
hist = model.fit(Xtr_scale, ytr, epochs=10, batch_size=100,␣
 ↪validation_data=(Xts_scale,yts))
```

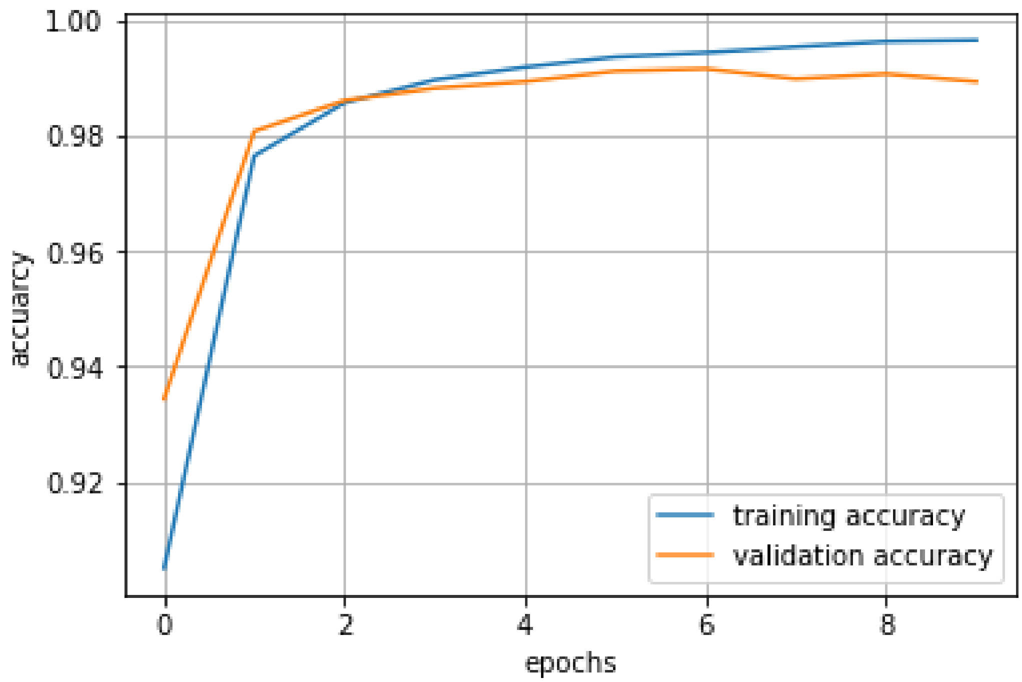
```
Train on 66247 samples, validate on 14904 samples
Epoch 1/10
66247/66247 [==============================] - 3s 51us/sample - loss: 0.3514 -
accuracy: 0.9051 - val_loss: 0.2037 - val_accuracy: 0.9344
Epoch 2/10
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0990 -
accuracy: 0.9766 - val_loss: 0.0865 - val_accuracy: 0.9809
Epoch 3/10
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0585 -
accuracy: 0.9858 - val_loss: 0.0597 - val_accuracy: 0.9862
Epoch 4/10
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0413 -
accuracy: 0.9898 - val_loss: 0.0484 - val_accuracy: 0.9883
Epoch 5/10
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0314 -
accuracy: 0.9920 - val_loss: 0.0369 - val_accuracy: 0.9895
Epoch 6/10
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0249 -
accuracy: 0.9937 - val_loss: 0.0316 - val_accuracy: 0.9913
Epoch 7/10
66247/66247 [==============================] - 3s 39us/sample - loss: 0.0205 -
accuracy: 0.9945 - val_loss: 0.0288 - val_accuracy: 0.9917
Epoch 8/10
66247/66247 [==============================] - 3s 38us/sample - loss: 0.0171 -
accuracy: 0.9955 - val_loss: 0.0324 - val_accuracy: 0.9899
Epoch 9/10
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0146 -
accuracy: 0.9964 - val_loss: 0.0313 - val_accuracy: 0.9908
Epoch 10/10
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0129 -
accuracy: 0.9966 - val_loss: 0.0320 - val_accuracy: 0.9895
```

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.

[20]:
```
# TODO 10
tr_accuracy = hist.history['accuracy']
val_accuracy = hist.history['val_accuracy']

plt.plot(tr_accuracy)
plt.plot(val_accuracy)
plt.grid()
plt.xlabel('epochs')
plt.ylabel('accuarcy')
plt.legend(['training accuracy', 'validation accuracy'])
```

[20]: <matplotlib.legend.Legend at 0x7fe080208190>



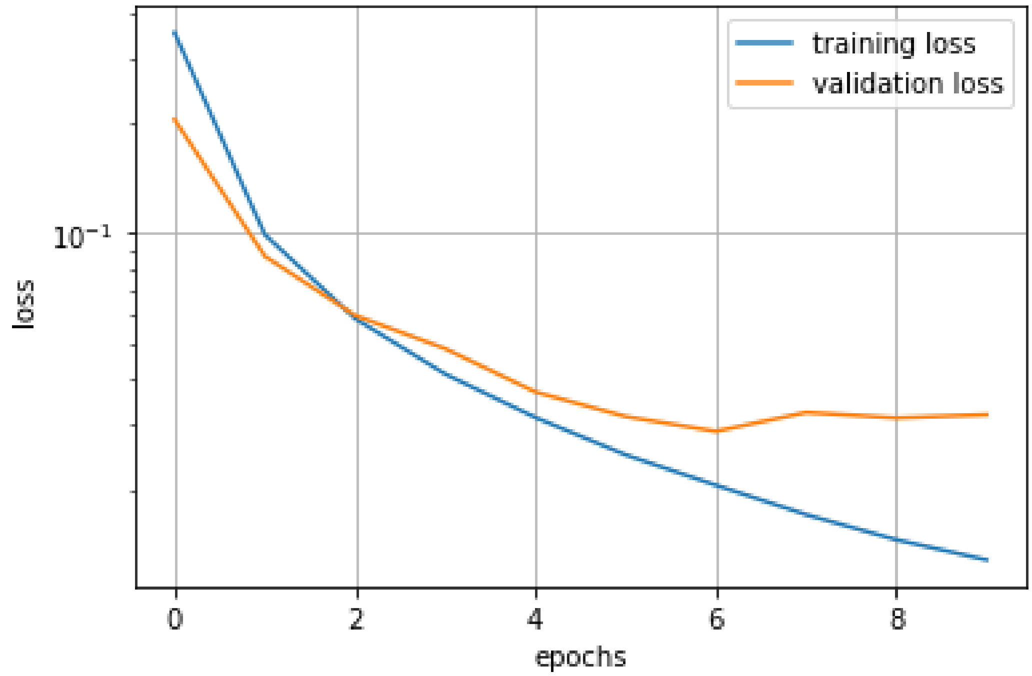
Plot the loss values saved in the `hist.history` dictionary. You should see that the loss is steadily decreasing. Use the `semilogy` plot.

[21]:
```
# TODO 11
tr_loss = hist.history['loss']
val_loss = hist.history['val_loss']

plt.semilogy(tr_loss)
plt.semilogy(val_loss)
plt.grid()
plt.xlabel('epochs')
plt.ylabel('loss')
```

```
plt.legend(['training loss', 'validation loss'])
```

[21]: <matplotlib.legend.Legend at 0x7fe0a88b4910>



1.5 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

[22]:
```
rates = [0.01,0.001,0.0001]
batch_size = 100
loss_hist_tr = []
loss_hist_val = []
acc_hist_tr = []
acc_hist_val = []

# TODO 12
for lr in rates:
    K.clear_session()
    model = Sequential()
    model.add(Dense(units=nh, input_shape=(nin,), activation='sigmoid',␣
 ↪name='hidden'))
    model.add(Dense(units=nout, activation='softmax', name='output'))
```

```
    opt = optimizers.Adam(lr=lr)
    model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
    hist = model.fit(Xtr_scale, ytr, epochs=20, batch_size=batch_size,␣
↪validation_data=(Xts_scale,yts))
    loss_hist_tr.append(hist.history['loss'])
    loss_hist_val.append(hist.history['val_loss'])
    acc_hist_tr.append(hist.history['accuracy'])
    acc_hist_val.append(hist.history['val_accuracy'])
```

```
Train on 66247 samples, validate on 14904 samples
Epoch 1/20
66247/66247 [==============================] - 3s 47us/sample - loss: 0.1061 -
accuracy: 0.9671 - val_loss: 0.0403 - val_accuracy: 0.9875
Epoch 2/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0299 -
accuracy: 0.9904 - val_loss: 0.0482 - val_accuracy: 0.9811
Epoch 3/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0212 -
accuracy: 0.9930 - val_loss: 0.1433 - val_accuracy: 0.9585
Epoch 4/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0188 -
accuracy: 0.9939 - val_loss: 0.1255 - val_accuracy: 0.9701
Epoch 5/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0170 -
accuracy: 0.9944 - val_loss: 0.0294 - val_accuracy: 0.9895
Epoch 6/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0134 -
accuracy: 0.9955 - val_loss: 0.0214 - val_accuracy: 0.9923
Epoch 7/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0119 -
accuracy: 0.9962 - val_loss: 0.0348 - val_accuracy: 0.9909
Epoch 8/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0135 -
accuracy: 0.9955 - val_loss: 0.0404 - val_accuracy: 0.9896
Epoch 9/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0116 -
accuracy: 0.9965 - val_loss: 0.0532 - val_accuracy: 0.9844
Epoch 10/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0155 -
accuracy: 0.9951 - val_loss: 0.0305 - val_accuracy: 0.9921
Epoch 11/20
66247/66247 [==============================] - 3s 39us/sample - loss: 0.0085 -
accuracy: 0.9973 - val_loss: 0.0248 - val_accuracy: 0.9928
Epoch 12/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0083 -
```

```
accuracy: 0.9975 - val_loss: 0.0625 - val_accuracy: 0.9820
Epoch 13/20
66247/66247 [==============================] - 3s 39us/sample - loss: 0.0084 -
accuracy: 0.9976 - val_loss: 0.0991 - val_accuracy: 0.9784
Epoch 14/20
66247/66247 [==============================] - 3s 39us/sample - loss: 0.0093 -
accuracy: 0.9969 - val_loss: 0.0508 - val_accuracy: 0.9903
Epoch 15/20
66247/66247 [==============================] - 3s 39us/sample - loss: 0.0096 -
accuracy: 0.9972 - val_loss: 0.0482 - val_accuracy: 0.9869
Epoch 16/20
66247/66247 [==============================] - 3s 38us/sample - loss: 0.0077 -
accuracy: 0.9977 - val_loss: 0.0460 - val_accuracy: 0.9889
Epoch 17/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0091 -
accuracy: 0.9973 - val_loss: 0.0376 - val_accuracy: 0.9908
Epoch 18/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0059 -
accuracy: 0.9981 - val_loss: 0.1059 - val_accuracy: 0.9812
Epoch 19/20
66247/66247 [==============================] - 3s 39us/sample - loss: 0.0109 -
accuracy: 0.9970 - val_loss: 0.0522 - val_accuracy: 0.9881
Epoch 20/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0077 -
accuracy: 0.9977 - val_loss: 0.0460 - val_accuracy: 0.9893
Train on 66247 samples, validate on 14904 samples
Epoch 1/20
66247/66247 [==============================] - 3s 47us/sample - loss: 0.3634 -
accuracy: 0.9025 - val_loss: 0.1992 - val_accuracy: 0.9409
Epoch 2/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.1030 -
accuracy: 0.9752 - val_loss: 0.1087 - val_accuracy: 0.9642
Epoch 3/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0600 -
accuracy: 0.9855 - val_loss: 0.0568 - val_accuracy: 0.9882
Epoch 4/20
66247/66247 [==============================] - 4s 54us/sample - loss: 0.0425 -
accuracy: 0.9892 - val_loss: 0.0594 - val_accuracy: 0.9809
Epoch 5/20
66247/66247 [==============================] - 3s 44us/sample - loss: 0.0320 -
accuracy: 0.9917 - val_loss: 0.0394 - val_accuracy: 0.9899
Epoch 6/20
66247/66247 [==============================] - 3s 44us/sample - loss: 0.0253 -
accuracy: 0.9933 - val_loss: 0.0323 - val_accuracy: 0.9905
Epoch 7/20
66247/66247 [==============================] - 3s 44us/sample - loss: 0.0207 -
accuracy: 0.9944 - val_loss: 0.0297 - val_accuracy: 0.9909
Epoch 8/20
```

```
66247/66247 [==============================] - 3s 47us/sample - loss: 0.0175 -
accuracy: 0.9955 - val_loss: 0.0245 - val_accuracy: 0.9932
Epoch 9/20
66247/66247 [==============================] - 3s 46us/sample - loss: 0.0149 -
accuracy: 0.9961 - val_loss: 0.0247 - val_accuracy: 0.9927
Epoch 10/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0129 -
accuracy: 0.9966 - val_loss: 0.0259 - val_accuracy: 0.9910
Epoch 11/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0112 -
accuracy: 0.9972 - val_loss: 0.0257 - val_accuracy: 0.9909
Epoch 12/20
66247/66247 [==============================] - 3s 43us/sample - loss: 0.0105 -
accuracy: 0.9974 - val_loss: 0.0263 - val_accuracy: 0.9908
Epoch 13/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0092 -
accuracy: 0.9976 - val_loss: 0.0240 - val_accuracy: 0.9913
Epoch 14/20
66247/66247 [==============================] - 3s 42us/sample - loss: 0.0082 -
accuracy: 0.9979 - val_loss: 0.0184 - val_accuracy: 0.9938
Epoch 15/20
66247/66247 [==============================] - 3s 42us/sample - loss: 0.0073 -
accuracy: 0.9982 - val_loss: 0.0195 - val_accuracy: 0.9926
Epoch 16/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0069 -
accuracy: 0.9982 - val_loss: 0.0239 - val_accuracy: 0.9910
Epoch 17/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0064 -
accuracy: 0.9984 - val_loss: 0.0190 - val_accuracy: 0.9927
Epoch 18/20
66247/66247 [==============================] - 3s 42us/sample - loss: 0.0057 -
accuracy: 0.9985 - val_loss: 0.0219 - val_accuracy: 0.9916
Epoch 19/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0054 -
accuracy: 0.9986 - val_loss: 0.0179 - val_accuracy: 0.9938
Epoch 20/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0050 -
accuracy: 0.9987 - val_loss: 0.0233 - val_accuracy: 0.9915
Train on 66247 samples, validate on 14904 samples
Epoch 1/20
66247/66247 [==============================] - 3s 51us/sample - loss: 1.1242 -
accuracy: 0.6527 - val_loss: 0.8697 - val_accuracy: 0.6672
Epoch 2/20
66247/66247 [==============================] - 3s 42us/sample - loss: 0.5620 -
accuracy: 0.8433 - val_loss: 0.5850 - val_accuracy: 0.8118
Epoch 3/20
66247/66247 [==============================] - 3s 42us/sample - loss: 0.3898 -
accuracy: 0.9082 - val_loss: 0.4579 - val_accuracy: 0.8610
```

```
Epoch 4/20
66247/66247 [==============================] - 3s 42us/sample - loss: 0.3011 -
accuracy: 0.9320 - val_loss: 0.3581 - val_accuracy: 0.8986
Epoch 5/20
66247/66247 [==============================] - 3s 42us/sample - loss: 0.2446 -
accuracy: 0.9456 - val_loss: 0.2884 - val_accuracy: 0.9244
Epoch 6/20
66247/66247 [==============================] - 3s 44us/sample - loss: 0.2040 -
accuracy: 0.9544 - val_loss: 0.2496 - val_accuracy: 0.9290
Epoch 7/20
66247/66247 [==============================] - 3s 49us/sample - loss: 0.1730 -
accuracy: 0.9603 - val_loss: 0.2143 - val_accuracy: 0.9383
Epoch 8/20
66247/66247 [==============================] - 3s 42us/sample - loss: 0.1487 -
accuracy: 0.9654 - val_loss: 0.1786 - val_accuracy: 0.9508
Epoch 9/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.1289 -
accuracy: 0.9697 - val_loss: 0.1517 - val_accuracy: 0.9628
Epoch 10/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.1130 -
accuracy: 0.9736 - val_loss: 0.1331 - val_accuracy: 0.9662
Epoch 11/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.1001 -
accuracy: 0.9769 - val_loss: 0.1275 - val_accuracy: 0.9632
Epoch 12/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0894 -
accuracy: 0.9796 - val_loss: 0.1068 - val_accuracy: 0.9731
Epoch 13/20
66247/66247 [==============================] - 3s 41us/sample - loss: 0.0806 -
accuracy: 0.9813 - val_loss: 0.0992 - val_accuracy: 0.9733
Epoch 14/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0733 -
accuracy: 0.9831 - val_loss: 0.0907 - val_accuracy: 0.9764
Epoch 15/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0672 -
accuracy: 0.9841 - val_loss: 0.0824 - val_accuracy: 0.9802
Epoch 16/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0619 -
accuracy: 0.9851 - val_loss: 0.0774 - val_accuracy: 0.9800
Epoch 17/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0574 -
accuracy: 0.9863 - val_loss: 0.0731 - val_accuracy: 0.9803
Epoch 18/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0535 -
accuracy: 0.9869 - val_loss: 0.0675 - val_accuracy: 0.9843
Epoch 19/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0501 -
accuracy: 0.9878 - val_loss: 0.0644 - val_accuracy: 0.9843
```

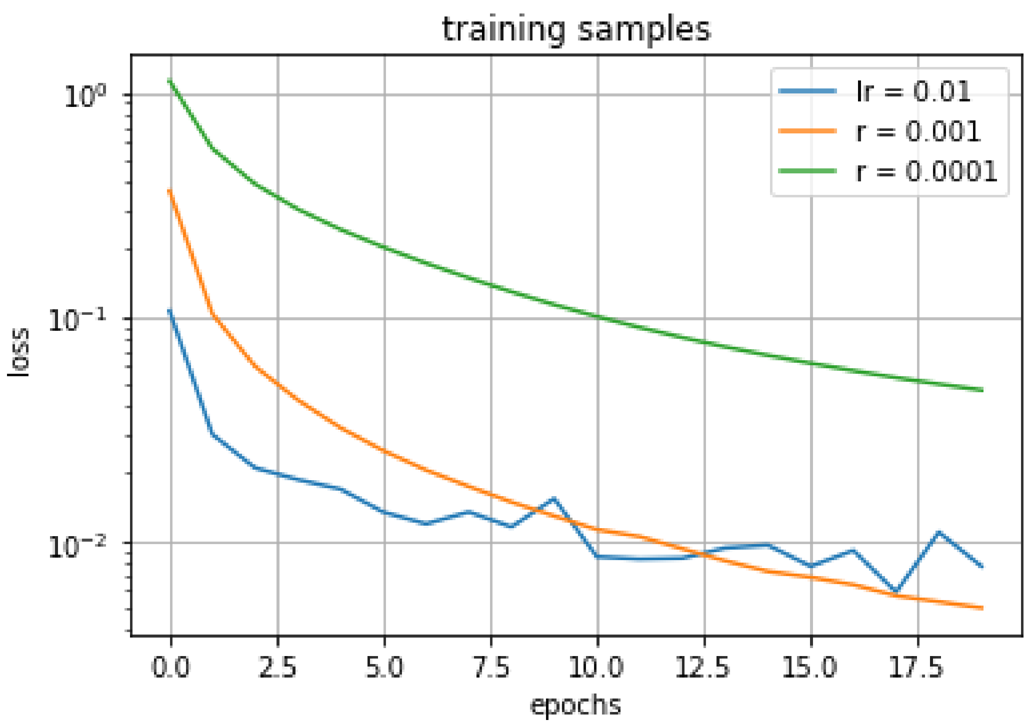
```
Epoch 20/20
66247/66247 [==============================] - 3s 40us/sample - loss: 0.0471 -
accuracy: 0.9883 - val_loss: 0.0598 - val_accuracy: 0.9859
```

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.

[23]:
```
# TODO 13
```

[24]:
```
plt.semilogy(loss_hist_tr[0], label = 'lr = 0.01')
plt.semilogy(loss_hist_tr[1], label = 'r = 0.001')
plt.semilogy(loss_hist_tr[2], label = 'r = 0.0001')
plt.grid()
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend()
plt.title('training samples')
```

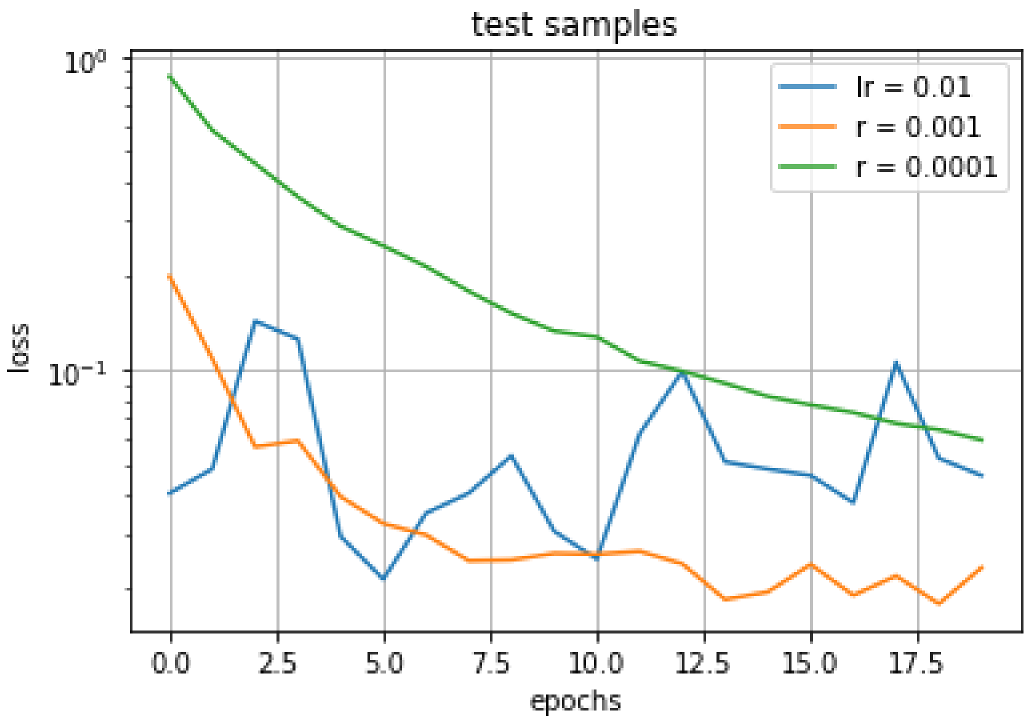
[24]:
```
Text(0.5, 1.0, 'training samples')
```



[25]:
```
plt.semilogy(loss_hist_val[0], label = 'lr = 0.01')
plt.semilogy(loss_hist_val[1], label = 'r = 0.001')
plt.semilogy(loss_hist_val[2], label = 'r = 0.0001')
plt.grid()
```

```
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend()
plt.title('test samples')
```

[25]: Text(0.5, 1.0, 'test samples')



[ ]: