lab4_music

June 3, 2021

1 Lab: Neural Networks for Music Classification

This lab is *optional* for undergraduate students, and *required* for graduate students. For all students, you may work in groups of one to four students.

In addition to the concepts in the MNIST demo posted in CCLE, 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/

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.

[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 -u 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

# TODO:  Load the file from url and save it in a file under the name fn
fn = "SopSax.Vib.pp.C6Eb6.aiff"
url = "http://theremin.music.uiowa.edu/sound files/MIS/Woodwinds/
 ↪sopranosaxophone/"+fn
```

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

[5]:
```python
# TODO
y, sr = librosa.load(fn)
```

You can listen to the audio being played with the `ipd.Audio` command. Set the `rate=sr`. You should hear the musician play four notes.

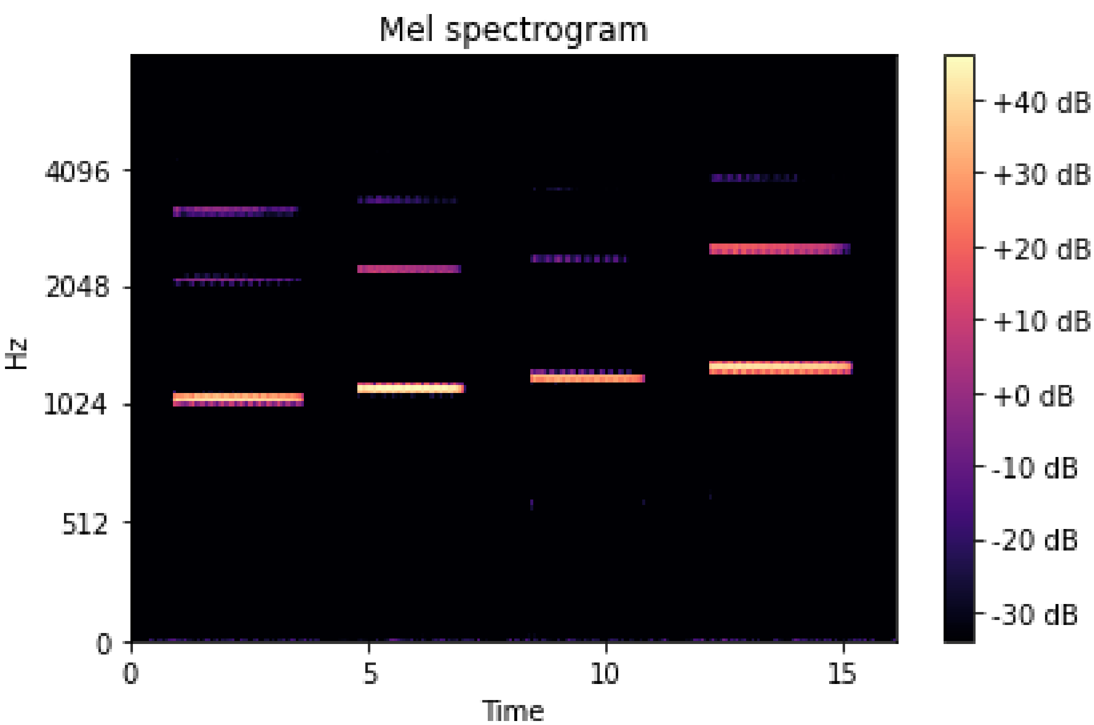
[6]:
```python
import IPython.display as ipd

# TODO
ipd.Audio(y, rate=sr)
```

[6]:
```
<IPython.lib.display.Audio object>
```

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
print('Number of training samples: ' + str(Xtr.shape[0]))
print('Number of test samples: ' + str(Xts.shape[0]))
print('Number of features: ' + str(Xtr.shape[1]))
print('Number of classes in each class: ' + str(np.unique(ytr).size))
```

```
Number of training samples: 66247
Number of test samples: 14904
Number of features: 120
Number of classes in each class: 10
```

Before continuing, you must scale the training and test data, `Xtr` and `Xts`. You can use the `StandardScaler` object which scales the data by removing the mean and dividing each feature by the standard deviation.

[10]:
```
from sklearn.preprocessing import StandardScaler

# TODO Scale the training and test matrices
sca = StandardScaler()
sca.fit(Xtr)
sca.fit(Xts)
Xtr_scale = sca.transform(Xtr)
Xts_scale = sca.transform(Xts)
```

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

[11]:
```
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Activation
```

[12]:
```
import tensorflow.keras.backend as K

# TODO clear session
K.clear_session()
```

[13]:
```
# TODO: construct the model
nin = 120
nh = 256
nout = 10

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

[14]:
```
# 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

[15]:
```
from tensorflow.keras import optimizers
import warnings
warnings.filterwarnings("ignore")

# TODO
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. Use a batch size of 100. Your final accuracy should be >99%.

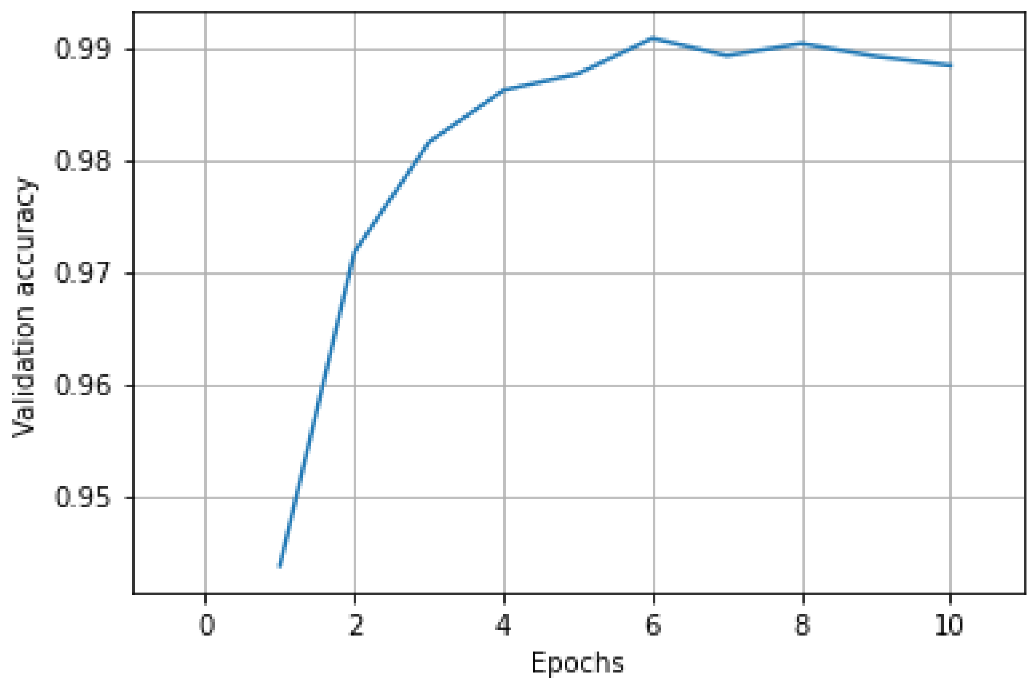
[16]:
```
# TODO
hist = model.fit(Xtr_scale, ytr, epochs=10, batch_size=100,␣
 ↪validation_data=(Xts_scale, yts))
```

```
Epoch 1/10
663/663 [==============================] - 1s 1ms/step - loss: 0.3588 -
accuracy: 0.9020 - val_loss: 0.1842 - val_accuracy: 0.9437
```

```
Epoch 2/10
663/663 [==============================] - 1s 923us/step - loss: 0.0981 -
accuracy: 0.9767 - val_loss: 0.0957 - val_accuracy: 0.9717
Epoch 3/10
663/663 [==============================] - 1s 925us/step - loss: 0.0574 -
accuracy: 0.9863 - val_loss: 0.0675 - val_accuracy: 0.9815
Epoch 4/10
663/663 [==============================] - 1s 928us/step - loss: 0.0400 -
accuracy: 0.9902 - val_loss: 0.0497 - val_accuracy: 0.9862
Epoch 5/10
663/663 [==============================] - 1s 929us/step - loss: 0.0297 -
accuracy: 0.9926 - val_loss: 0.0429 - val_accuracy: 0.9877
Epoch 6/10
663/663 [==============================] - 1s 933us/step - loss: 0.0234 -
accuracy: 0.9940 - val_loss: 0.0347 - val_accuracy: 0.9908
Epoch 7/10
663/663 [==============================] - 1s 952us/step - loss: 0.0190 -
accuracy: 0.9952 - val_loss: 0.0307 - val_accuracy: 0.9893
Epoch 8/10
663/663 [==============================] - 1s 997us/step - loss: 0.0159 -
accuracy: 0.9959 - val_loss: 0.0298 - val_accuracy: 0.9903
Epoch 9/10
663/663 [==============================] - 1s 954us/step - loss: 0.0134 -
accuracy: 0.9966 - val_loss: 0.0277 - val_accuracy: 0.9892
Epoch 10/10
663/663 [==============================] - 1s 969us/step - loss: 0.0119 -
accuracy: 0.9970 - val_loss: 0.0337 - val_accuracy: 0.9884
```

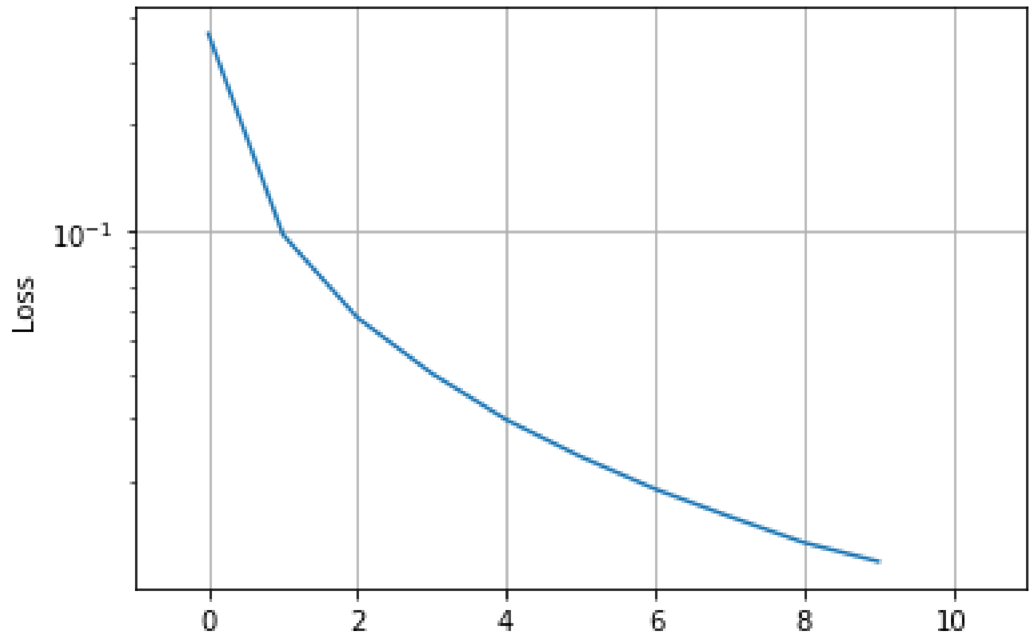
Plot the validation accuracy saved in `hist.history` dictionary. This gives one accuracy value per epoch. You should see that the validation accuracy saturates at a little higher than 99%. After that it "bounces around" due to the noise in the stochastic gradient descent.

[17]:
```
# TODO
epochs = np.arange(1,11)
plt.plot(epochs, hist.history['val_accuracy'])
plt.xlim(-1, 11)
plt.ylabel('Validation accuracy')
plt.xlabel('Epochs')
plt.grid()
plt.show()
```



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

[18]:
```
# TODO
histo = np.array(hist.history['loss'])
plt.semilogy(histo)
plt.xlim(-1, 11)
plt.ylabel('Loss')
plt.grid()
plt.show()
```



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

[19]:
```
rates = [0.01, 0.001, 0.0001]
loss_hist = []

# TODO
for lr in rates:

    # TODO:  Clear the session
    K.clear_session()

    # TODO:  Build the model
    nin = 120
    nh = 256
    nout = 10
    model = Sequential()
    model.add(Dense(units=nh, input_shape=(nin,), activation='sigmoid',␣
 ↪name='hidden'))
    model.add(Dense(units=nout, activation='softmax', name='output'))

    # TODO:  Select the optimizer with the correct learning rate to test
```

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

    # TODO:  Fit the model
    hist = model.fit(Xtr_scale, ytr, epochs=20, batch_size=100,␣
↪validation_data=(Xts_scale, yts))

    # TODO:  Save the loss history
    loss_hist.append(hist.history['loss'])

    # TODO:  Print the final accuracy
    print(hist.history['accuracy'])
```

```
Epoch 1/20
663/663 [==============================] - 1s 1ms/step - loss: 0.1042 -
accuracy: 0.9679 - val_loss: 0.0606 - val_accuracy: 0.9817
Epoch 2/20
663/663 [==============================] - 1s 971us/step - loss: 0.0301 -
accuracy: 0.9901 - val_loss: 0.0375 - val_accuracy: 0.9880
Epoch 3/20
663/663 [==============================] - 1s 927us/step - loss: 0.0216 -
accuracy: 0.9929 - val_loss: 0.0615 - val_accuracy: 0.9785
Epoch 4/20
663/663 [==============================] - 1s 929us/step - loss: 0.0200 -
accuracy: 0.9934 - val_loss: 0.0266 - val_accuracy: 0.9909
Epoch 5/20
663/663 [==============================] - 1s 957us/step - loss: 0.0149 -
accuracy: 0.9948 - val_loss: 0.0532 - val_accuracy: 0.9828
Epoch 6/20
663/663 [==============================] - 1s 962us/step - loss: 0.0164 -
accuracy: 0.9951 - val_loss: 0.0346 - val_accuracy: 0.9891
Epoch 7/20
663/663 [==============================] - 1s 981us/step - loss: 0.0116 -
accuracy: 0.9964 - val_loss: 0.0481 - val_accuracy: 0.9850
Epoch 8/20
663/663 [==============================] - 1s 959us/step - loss: 0.0128 -
accuracy: 0.9957 - val_loss: 0.1110 - val_accuracy: 0.9697
Epoch 9/20
663/663 [==============================] - 1s 964us/step - loss: 0.0134 -
accuracy: 0.9957 - val_loss: 0.0580 - val_accuracy: 0.9849
Epoch 10/20
663/663 [==============================] - 1s 965us/step - loss: 0.0087 -
accuracy: 0.9972 - val_loss: 0.0715 - val_accuracy: 0.9808
Epoch 11/20
663/663 [==============================] - 1s 960us/step - loss: 0.0115 -
accuracy: 0.9965 - val_loss: 0.0906 - val_accuracy: 0.9791
```

```
Epoch 12/20
663/663 [==============================] - 1s 966us/step - loss: 0.0115 -
accuracy: 0.9968 - val_loss: 0.1161 - val_accuracy: 0.9747
Epoch 13/20
663/663 [==============================] - 1s 971us/step - loss: 0.0103 -
accuracy: 0.9969 - val_loss: 0.0378 - val_accuracy: 0.9899
Epoch 14/20
663/663 [==============================] - 1s 960us/step - loss: 0.0108 -
accuracy: 0.9969 - val_loss: 0.0813 - val_accuracy: 0.9813
Epoch 15/20
663/663 [==============================] - 1s 1ms/step - loss: 0.0074 -
accuracy: 0.9978 - val_loss: 0.0578 - val_accuracy: 0.9856
Epoch 16/20
663/663 [==============================] - 1s 1ms/step - loss: 0.0104 -
accuracy: 0.9970 - val_loss: 0.0758 - val_accuracy: 0.9816
Epoch 17/20
663/663 [==============================] - 1s 995us/step - loss: 0.0078 -
accuracy: 0.9975 - val_loss: 0.0358 - val_accuracy: 0.9902
Epoch 18/20
663/663 [==============================] - 1s 962us/step - loss: 0.0081 -
accuracy: 0.9978 - val_loss: 0.0421 - val_accuracy: 0.9879
Epoch 19/20
663/663 [==============================] - 1s 959us/step - loss: 0.0089 -
accuracy: 0.9973 - val_loss: 0.0855 - val_accuracy: 0.9856
Epoch 20/20
663/663 [==============================] - 1s 963us/step - loss: 0.0083 -
accuracy: 0.9976 - val_loss: 0.1518 - val_accuracy: 0.9766
[0.9679079651832581, 0.9900524020195007, 0.9929355382919312, 0.9934185743331909,
0.9947771430015564, 0.9950790405273438, 0.9963923096656799, 0.9957129955291748,
0.9956526160240173, 0.9972225427627563, 0.9965130686759949, 0.9967696666717529,
0.9968904256820679, 0.9968904256820679, 0.997826337814331, 0.9969508051872253,
0.9974640607833862, 0.9977810382843018, 0.9973130822181702, 0.9975696802139282]
Epoch 1/20
663/663 [==============================] - 1s 1ms/step - loss: 0.3406 -
accuracy: 0.9082 - val_loss: 0.1620 - val_accuracy: 0.9655
Epoch 2/20
663/663 [==============================] - 1s 1ms/step - loss: 0.0978 -
accuracy: 0.9770 - val_loss: 0.0908 - val_accuracy: 0.9750
Epoch 3/20
663/663 [==============================] - 1s 993us/step - loss: 0.0573 -
accuracy: 0.9864 - val_loss: 0.0601 - val_accuracy: 0.9841
Epoch 4/20
663/663 [==============================] - 1s 996us/step - loss: 0.0395 -
accuracy: 0.9907 - val_loss: 0.0481 - val_accuracy: 0.9864
Epoch 5/20
663/663 [==============================] - 1s 984us/step - loss: 0.0297 -
accuracy: 0.9926 - val_loss: 0.0398 - val_accuracy: 0.9889
Epoch 6/20
```

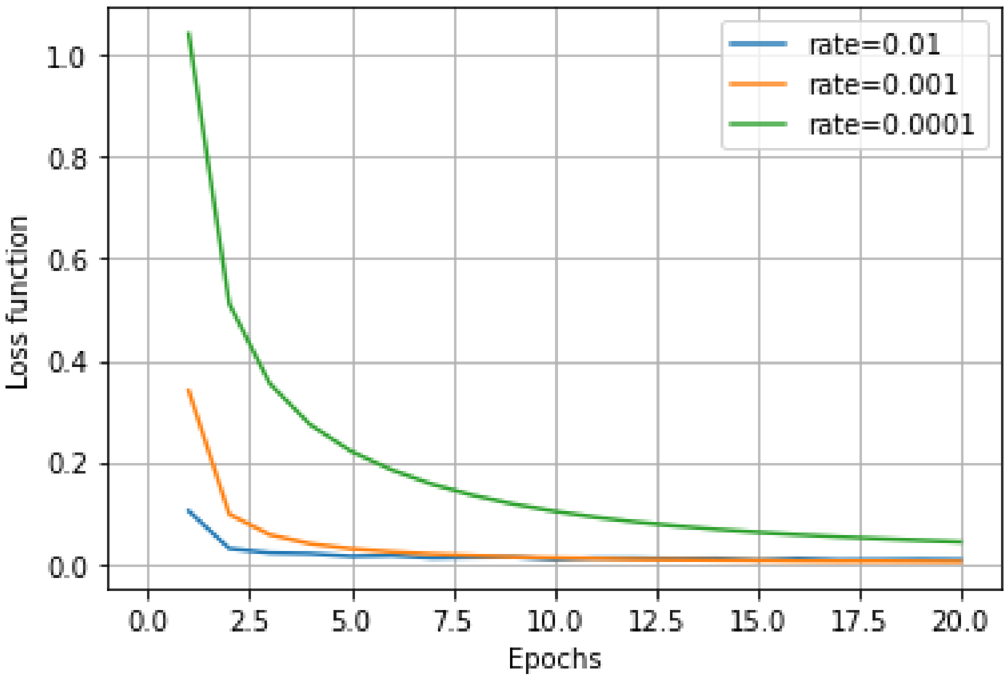
```
663/663 [==============================] - 1s 952us/step - loss: 0.0236 -
accuracy: 0.9940 - val_loss: 0.0391 - val_accuracy: 0.9872
Epoch 7/20
663/663 [==============================] - 1s 950us/step - loss: 0.0191 -
accuracy: 0.9952 - val_loss: 0.0309 - val_accuracy: 0.9905
Epoch 8/20
663/663 [==============================] - 1s 953us/step - loss: 0.0163 -
accuracy: 0.9960 - val_loss: 0.0307 - val_accuracy: 0.9907
Epoch 9/20
663/663 [==============================] - 1s 958us/step - loss: 0.0135 -
accuracy: 0.9968 - val_loss: 0.0278 - val_accuracy: 0.9908
Epoch 10/20
663/663 [==============================] - 1s 950us/step - loss: 0.0118 -
accuracy: 0.9971 - val_loss: 0.0265 - val_accuracy: 0.9908
Epoch 11/20
663/663 [==============================] - 1s 957us/step - loss: 0.0106 -
accuracy: 0.9973 - val_loss: 0.0274 - val_accuracy: 0.9901
Epoch 12/20
663/663 [==============================] - 1s 981us/step - loss: 0.0089 -
accuracy: 0.9979 - val_loss: 0.0233 - val_accuracy: 0.9918
Epoch 13/20
663/663 [==============================] - 1s 985us/step - loss: 0.0083 -
accuracy: 0.9979 - val_loss: 0.0261 - val_accuracy: 0.9902
Epoch 14/20
663/663 [==============================] - 1s 1ms/step - loss: 0.0074 -
accuracy: 0.9979 - val_loss: 0.0232 - val_accuracy: 0.9921
Epoch 15/20
663/663 [==============================] - 1s 1ms/step - loss: 0.0068 -
accuracy: 0.9981 - val_loss: 0.0220 - val_accuracy: 0.9917
Epoch 16/20
663/663 [==============================] - 1s 1ms/step - loss: 0.0059 -
accuracy: 0.9985 - val_loss: 0.0229 - val_accuracy: 0.9920
Epoch 17/20
663/663 [==============================] - 1s 1ms/step - loss: 0.0054 -
accuracy: 0.9987 - val_loss: 0.0243 - val_accuracy: 0.9908
Epoch 18/20
663/663 [==============================] - 1s 1ms/step - loss: 0.0053 -
accuracy: 0.9987 - val_loss: 0.0207 - val_accuracy: 0.9932
Epoch 19/20
663/663 [==============================] - 1s 1ms/step - loss: 0.0048 -
accuracy: 0.9988 - val_loss: 0.0215 - val_accuracy: 0.9928
Epoch 20/20
663/663 [==============================] - 1s 1ms/step - loss: 0.0044 -
accuracy: 0.9989 - val_loss: 0.0203 - val_accuracy: 0.9924
[0.9082071781158447, 0.9769800901412964, 0.9863842725753784, 0.9906863570213318,
0.992648720741272, 0.9939619898796082, 0.9951545000076294, 0.9959545135498047,
0.9968300461769104, 0.9970564842224121, 0.9972677826881409, 0.9978564977645874,
0.9979168772697449, 0.9979168772697449, 0.9980980157852173, 0.9985055923461914,
```

```
0.9986565709114075, 0.9986716508865356, 0.9987924098968506, 0.9989433288574219]
Epoch 1/20
663/663 [==============================] - 1s 1ms/step - loss: 1.0420 -
accuracy: 0.6836 - val_loss: 0.7755 - val_accuracy: 0.7203
Epoch 2/20
663/663 [==============================] - 1s 995us/step - loss: 0.5112 -
accuracy: 0.8658 - val_loss: 0.5228 - val_accuracy: 0.8486
Epoch 3/20
663/663 [==============================] - 1s 1ms/step - loss: 0.3544 -
accuracy: 0.9196 - val_loss: 0.3934 - val_accuracy: 0.8940
Epoch 4/20
663/663 [==============================] - 1s 954us/step - loss: 0.2733 -
accuracy: 0.9391 - val_loss: 0.3130 - val_accuracy: 0.9141
Epoch 5/20
663/663 [==============================] - 1s 957us/step - loss: 0.2214 -
accuracy: 0.9500 - val_loss: 0.2594 - val_accuracy: 0.9273
Epoch 6/20
663/663 [==============================] - 1s 961us/step - loss: 0.1845 -
accuracy: 0.9582 - val_loss: 0.2143 - val_accuracy: 0.9420
Epoch 7/20
663/663 [==============================] - 1s 984us/step - loss: 0.1566 -
accuracy: 0.9644 - val_loss: 0.1831 - val_accuracy: 0.9491
Epoch 8/20
663/663 [==============================] - 1s 969us/step - loss: 0.1349 -
accuracy: 0.9696 - val_loss: 0.1633 - val_accuracy: 0.9525
Epoch 9/20
663/663 [==============================] - 1s 1ms/step - loss: 0.1176 -
accuracy: 0.9728 - val_loss: 0.1399 - val_accuracy: 0.9619
Epoch 10/20
663/663 [==============================] - 1s 1ms/step - loss: 0.1036 -
accuracy: 0.9762 - val_loss: 0.1225 - val_accuracy: 0.9677
Epoch 11/20
663/663 [==============================] - 1s 980us/step - loss: 0.0920 -
accuracy: 0.9789 - val_loss: 0.1077 - val_accuracy: 0.9734
Epoch 12/20
663/663 [==============================] - 1s 949us/step - loss: 0.0825 -
accuracy: 0.9811 - val_loss: 0.0940 - val_accuracy: 0.9783
Epoch 13/20
663/663 [==============================] - 1s 967us/step - loss: 0.0746 -
accuracy: 0.9829 - val_loss: 0.0918 - val_accuracy: 0.9750
Epoch 14/20
663/663 [==============================] - 1s 969us/step - loss: 0.0679 -
accuracy: 0.9845 - val_loss: 0.0822 - val_accuracy: 0.9789
Epoch 15/20
663/663 [==============================] - 1s 970us/step - loss: 0.0622 -
accuracy: 0.9857 - val_loss: 0.0778 - val_accuracy: 0.9797
Epoch 16/20
663/663 [==============================] - 1s 977us/step - loss: 0.0574 -
```

```
accuracy: 0.9868 - val_loss: 0.0689 - val_accuracy: 0.9832
Epoch 17/20
663/663 [==============================] - 1s 972us/step - loss: 0.0531 -
accuracy: 0.9878 - val_loss: 0.0660 - val_accuracy: 0.9833
Epoch 18/20
663/663 [==============================] - 1s 979us/step - loss: 0.0495 -
accuracy: 0.9885 - val_loss: 0.0623 - val_accuracy: 0.9842
Epoch 19/20
663/663 [==============================] - 1s 994us/step - loss: 0.0463 -
accuracy: 0.9892 - val_loss: 0.0587 - val_accuracy: 0.9854
Epoch 20/20
663/663 [==============================] - 1s 978us/step - loss: 0.0433 -
accuracy: 0.9898 - val_loss: 0.0565 - val_accuracy: 0.9854
[0.6835781335830688, 0.8658052682876587, 0.9195888042449951, 0.9390764832496643,
0.9500354528427124, 0.9581716656684875, 0.9644361138343811, 0.9696288108825684,
0.972798764705658, 0.9761951565742493, 0.9789273738861084, 0.981055736541748,
0.9828822612762451, 0.9844822883605957, 0.9857050180435181, 0.9868220686912537,
0.9878334403038025, 0.9884523153305054, 0.989176869392395, 0.9897505044937134]
```

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.

[20]:
```
# TODO
loss_hist = np.array(loss_hist)
epochs = np.arange(1, 21)

plt.plot(epochs, loss_hist[0, :], label='rate=0.01')
plt.plot(epochs, loss_hist[1, :], label='rate=0.001')
plt.plot(epochs, loss_hist[2, :], label='rate=0.0001')
plt.legend()
plt.xlim(-1, 21)
plt.xlabel('Epochs')
plt.ylabel('Loss function')
plt.grid()
plt.show()
```


rate=0.01
rate=0.001
rate=0.0001
Loss function
Epochs
1.0
0.8
0.6
0.4
0.2
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0