Copy_of_lab_music_partial

April 15, 2022

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.

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

Then, load the other packages.

```
[ ]: 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.

```
[ ]: 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).

```
[ ]: 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
     r = requests.get(url)
     open(fn, 'wb').write(r.content)
```

```
[ ]: 1418242
```

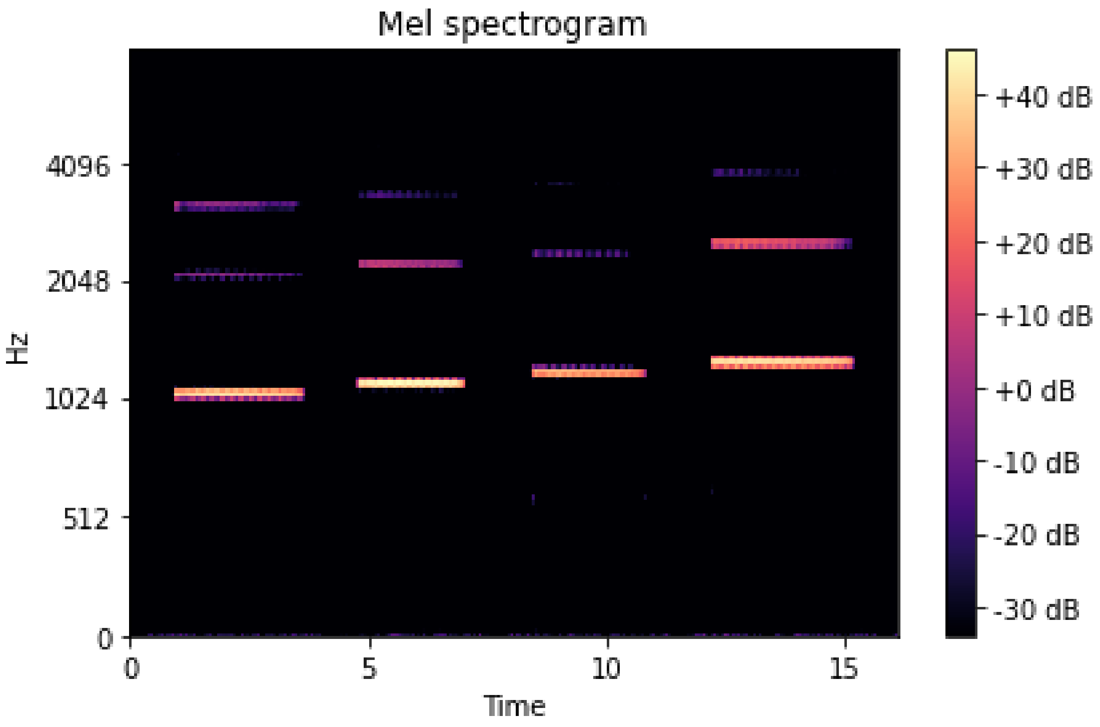
Next, use `librosa` command `librosa.load` to read the audio file with filename `fn` and get the samples `y` and sample rate `sr`.

```
[ ]: # TODO 2
     # 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.

[ ]:
```
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.

[69]:
```
Xtr = np.load('instrument_dataset/uiowa_train_data.npy')
ytr = np.load('instrument_dataset/uiowa_train_labels.npy')
Xts = np.load('instrument_dataset/uiowa_test_data.npy')
```

```
yts = np.load('instrument_dataset/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?

[71]:
```
# TODO 3
print('The number of training and test samples are '+str(Xtr.shape[0])
+' and '+str(Xts.shape[0]))
print('The number of features for each sample = '+str(Xtr.shape[1]))
print('There are '+str(np.unique(yts).shape[0])+' instruments per class')
```

```
The number of training and test samples are 66247 and 14904
The number of features for each sample = 120
There are 10 instruments per class
```

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.

[72]:
```
# TODO 4: Scale the training and test matrices
# Xtr_scale = ...
# Xts_scale = ...
from sklearn.preprocessing import StandardScaler
scaler1 = StandardScaler(with_mean=True, with_std=True)
scaler2 = StandardScaler(with_mean=True, with_std=True)
Xtr_scale = scaler1.fit_transform(Xtr)
Xts_scale = scaler2.fit_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

[73]:
```
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Activation
import tensorflow.keras.backend as K
```

[74]:
```
# TODO 5 clear session
K.clear_session()
```

[76]:
```
# TODO 6: construct the model
# TODO 6: construct the model
nin = Xtr.shape[1] # dimension of input data
nh = 256 # number of hidden units
nout = int(np.max(ytr)+1) # number of outputs = 10 since there are 10␣
 ↪instruments
model = Sequential()
```

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

[77]:
```
# 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

[80]:
```
# TODO 8
# opt = ...
# model.compile(...)
from tensorflow.keras import optimizers
opt = optimizers.Adam(lr=0.001)
model.
 ↪compile(optimizer=opt,loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/adam.py:105:
UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)
```

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%.

[81]:
```
# TODO 9
hist = model.fit(Xtr_scale, ytr, epochs=10, batch_size=100,
validation_data=(Xts_scale,yts))
```

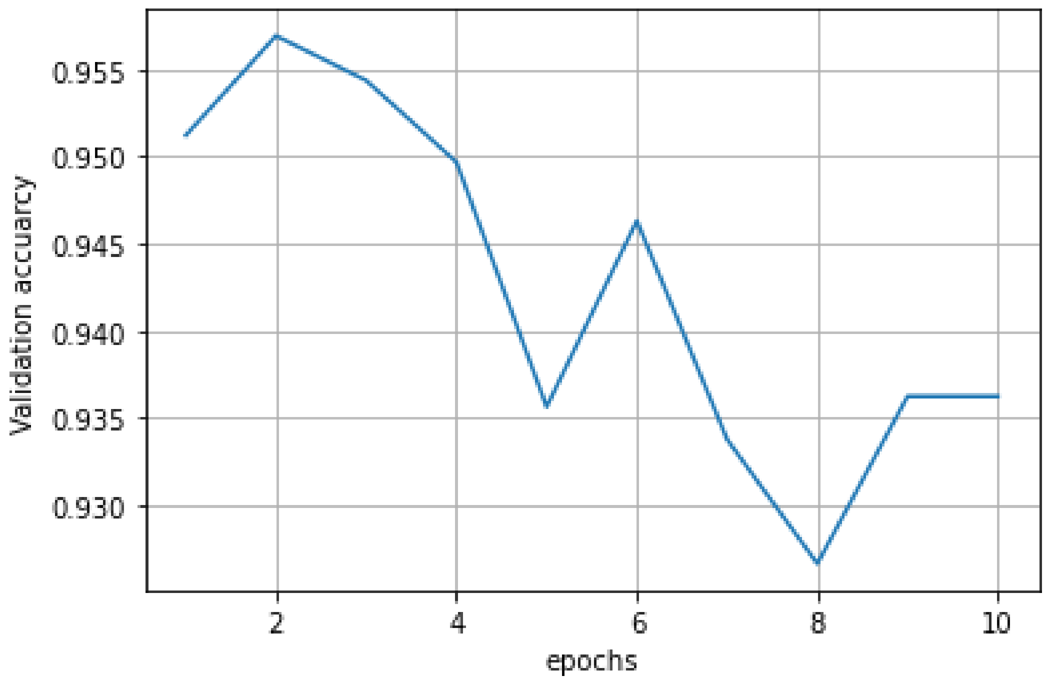
```
Epoch 1/10
663/663 [==============================] - 2s 3ms/step - loss: 0.3773 -
accuracy: 0.8973 - val_loss: 0.1952 - val_accuracy: 0.9512
Epoch 2/10
663/663 [==============================] - 2s 3ms/step - loss: 0.1065 -
accuracy: 0.9738 - val_loss: 0.1371 - val_accuracy: 0.9569
```

```
Epoch 3/10
663/663 [==============================] - 2s 3ms/step - loss: 0.0625 -
accuracy: 0.9851 - val_loss: 0.1250 - val_accuracy: 0.9544
Epoch 4/10
663/663 [==============================] - 2s 3ms/step - loss: 0.0430 -
accuracy: 0.9895 - val_loss: 0.1379 - val_accuracy: 0.9497
Epoch 5/10
663/663 [==============================] - 2s 4ms/step - loss: 0.0328 -
accuracy: 0.9914 - val_loss: 0.1604 - val_accuracy: 0.9357
Epoch 6/10
663/663 [==============================] - 2s 3ms/step - loss: 0.0259 -
accuracy: 0.9934 - val_loss: 0.1399 - val_accuracy: 0.9463
Epoch 7/10
663/663 [==============================] - 2s 2ms/step - loss: 0.0211 -
accuracy: 0.9947 - val_loss: 0.1726 - val_accuracy: 0.9338
Epoch 8/10
663/663 [==============================] - 2s 3ms/step - loss: 0.0177 -
accuracy: 0.9954 - val_loss: 0.1951 - val_accuracy: 0.9267
Epoch 9/10
663/663 [==============================] - 2s 4ms/step - loss: 0.0151 -
accuracy: 0.9958 - val_loss: 0.1747 - val_accuracy: 0.9363
Epoch 10/10
663/663 [==============================] - 4s 6ms/step - loss: 0.0129 -
accuracy: 0.9966 - val_loss: 0.1827 - val_accuracy: 0.9363
```

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.

[82]:
```
# TODO 10
val_accuracy = hist.history['val_accuracy']
epochs=np.arange(start=1, stop=11)
plt.plot(epochs,val_accuracy)
plt.grid()
plt.xlabel('epochs')
plt.ylabel('Validation accuarcy')
```

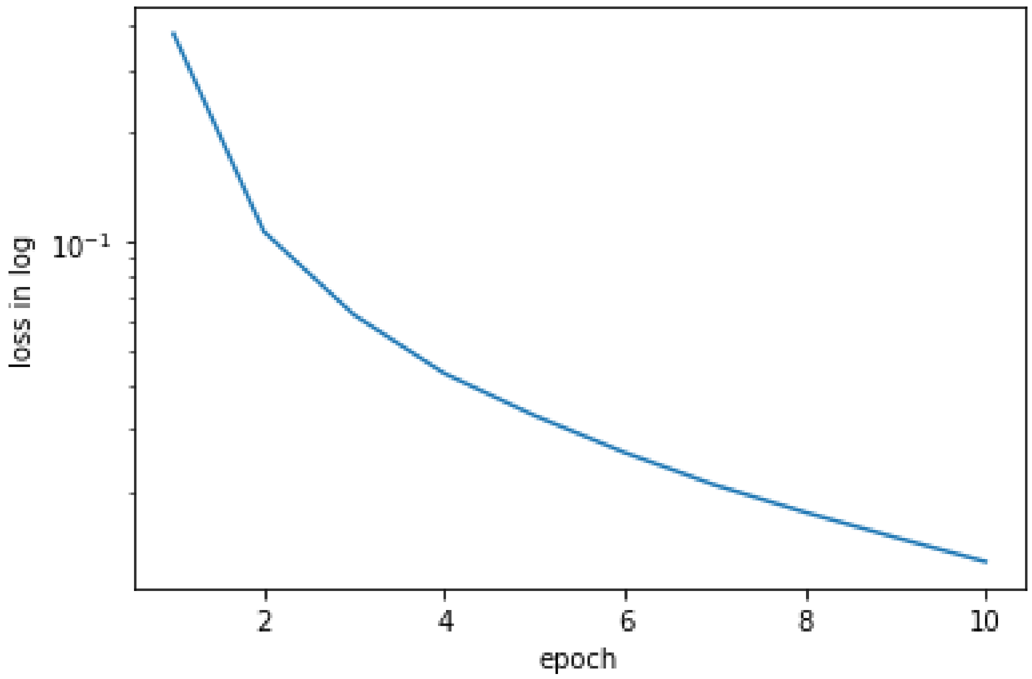
[82]:
```
Text(0, 0.5, 'Validation accuarcy')
```



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

[84]:
```
# TODO 11
loss = hist.history['loss']
epochs=np.arange(start=1, stop=11)
plt.semilogy(epochs,loss)
plt.ylabel('loss in log')
plt.xlabel('epoch')
```

[84]:
```
Text(0.5, 0, 'epoch')
```



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

[87]:
```
rates = [0.01,0.001,0.0001]
batch_size = 100
loss_hist = []
tracc_hist = []
vacc_hist = []

# TODO 12
# for lr in rate:
#     ...
i=0
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'])
  hist1 = model.fit(Xtr_scale, ytr, epochs=20,␣
 ↪batch_size=100,validation_data=(Xts_scale,yts))
  loss_hist.append(hist1.history['loss'])
  tracc_hist.append(hist1.history['accuracy'])
  vacc_hist.append(hist1.history['val_accuracy'])
  i+=1
```

```
Epoch 1/20

/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/adam.py:105:
UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)

663/663 [==============================] - 2s 3ms/step - loss: 0.1047 -
accuracy: 0.9680 - val_loss: 0.1561 - val_accuracy: 0.9472
Epoch 2/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0310 -
accuracy: 0.9897 - val_loss: 0.1159 - val_accuracy: 0.9625
Epoch 3/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0207 -
accuracy: 0.9928 - val_loss: 0.1577 - val_accuracy: 0.9542
Epoch 4/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0171 -
accuracy: 0.9942 - val_loss: 0.1644 - val_accuracy: 0.9551
Epoch 5/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0151 -
accuracy: 0.9952 - val_loss: 0.3947 - val_accuracy: 0.9216
Epoch 6/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0174 -
accuracy: 0.9943 - val_loss: 0.3069 - val_accuracy: 0.9363
Epoch 7/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0124 -
accuracy: 0.9960 - val_loss: 0.5620 - val_accuracy: 0.9070
Epoch 8/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0149 -
accuracy: 0.9955 - val_loss: 0.3751 - val_accuracy: 0.9282
Epoch 9/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0119 -
accuracy: 0.9962 - val_loss: 0.6034 - val_accuracy: 0.9073
Epoch 10/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0097 -
accuracy: 0.9969 - val_loss: 0.4097 - val_accuracy: 0.9287
Epoch 11/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0132 -
accuracy: 0.9962 - val_loss: 0.4222 - val_accuracy: 0.9273
Epoch 12/20
```

```
663/663 [==============================] - 2s 3ms/step - loss: 0.0111 -
accuracy: 0.9963 - val_loss: 0.7344 - val_accuracy: 0.8983
Epoch 13/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0109 -
accuracy: 0.9969 - val_loss: 0.4032 - val_accuracy: 0.9277
Epoch 14/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0104 -
accuracy: 0.9970 - val_loss: 1.2674 - val_accuracy: 0.8747
Epoch 15/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0092 -
accuracy: 0.9974 - val_loss: 0.7153 - val_accuracy: 0.9025
Epoch 16/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0095 -
accuracy: 0.9971 - val_loss: 0.8552 - val_accuracy: 0.8970
Epoch 17/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0107 -
accuracy: 0.9968 - val_loss: 0.5554 - val_accuracy: 0.9218
Epoch 18/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0051 -
accuracy: 0.9984 - val_loss: 0.9520 - val_accuracy: 0.8977
Epoch 19/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0089 -
accuracy: 0.9975 - val_loss: 1.8345 - val_accuracy: 0.8612
Epoch 20/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0105 -
accuracy: 0.9971 - val_loss: 0.9512 - val_accuracy: 0.8824
Epoch 1/20
663/663 [==============================] - 2s 3ms/step - loss: 0.3708 -
accuracy: 0.8999 - val_loss: 0.1858 - val_accuracy: 0.9573
Epoch 2/20
663/663 [==============================] - 2s 3ms/step - loss: 0.1049 -
accuracy: 0.9746 - val_loss: 0.1592 - val_accuracy: 0.9463
Epoch 3/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0623 -
accuracy: 0.9852 - val_loss: 0.1246 - val_accuracy: 0.9552
Epoch 4/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0440 -
accuracy: 0.9890 - val_loss: 0.1256 - val_accuracy: 0.9541
Epoch 5/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0333 -
accuracy: 0.9914 - val_loss: 0.1586 - val_accuracy: 0.9371
Epoch 6/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0262 -
accuracy: 0.9932 - val_loss: 0.1606 - val_accuracy: 0.9377
Epoch 7/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0214 -
accuracy: 0.9946 - val_loss: 0.1684 - val_accuracy: 0.9340
Epoch 8/20
```

```
663/663 [==============================] - 2s 3ms/step - loss: 0.0182 -
accuracy: 0.9951 - val_loss: 0.1611 - val_accuracy: 0.9418
Epoch 9/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0147 -
accuracy: 0.9963 - val_loss: 0.2562 - val_accuracy: 0.9145
Epoch 10/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0132 -
accuracy: 0.9965 - val_loss: 0.2086 - val_accuracy: 0.9308
Epoch 11/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0115 -
accuracy: 0.9970 - val_loss: 0.2234 - val_accuracy: 0.9273
Epoch 12/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0101 -
accuracy: 0.9974 - val_loss: 0.2211 - val_accuracy: 0.9304
Epoch 13/20
663/663 [==============================] - 2s 2ms/step - loss: 0.0091 -
accuracy: 0.9976 - val_loss: 0.3809 - val_accuracy: 0.9015
Epoch 14/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0083 -
accuracy: 0.9979 - val_loss: 0.3851 - val_accuracy: 0.9015
Epoch 15/20
663/663 [==============================] - 2s 2ms/step - loss: 0.0075 -
accuracy: 0.9980 - val_loss: 0.2379 - val_accuracy: 0.9324
Epoch 16/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0068 -
accuracy: 0.9982 - val_loss: 0.3340 - val_accuracy: 0.9159
Epoch 17/20
663/663 [==============================] - 2s 2ms/step - loss: 0.0064 -
accuracy: 0.9983 - val_loss: 0.3780 - val_accuracy: 0.9109
Epoch 18/20
663/663 [==============================] - 2s 4ms/step - loss: 0.0058 -
accuracy: 0.9986 - val_loss: 0.2990 - val_accuracy: 0.9233
Epoch 19/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0055 -
accuracy: 0.9985 - val_loss: 0.3825 - val_accuracy: 0.9145
Epoch 20/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0049 -
accuracy: 0.9987 - val_loss: 0.3734 - val_accuracy: 0.9156
Epoch 1/20
663/663 [==============================] - 2s 3ms/step - loss: 1.1021 -
accuracy: 0.6553 - val_loss: 0.7964 - val_accuracy: 0.7287
Epoch 2/20
663/663 [==============================] - 2s 3ms/step - loss: 0.5470 -
accuracy: 0.8561 - val_loss: 0.5267 - val_accuracy: 0.8598
Epoch 3/20
663/663 [==============================] - 2s 3ms/step - loss: 0.3806 -
accuracy: 0.9134 - val_loss: 0.3974 - val_accuracy: 0.9044
Epoch 4/20
```

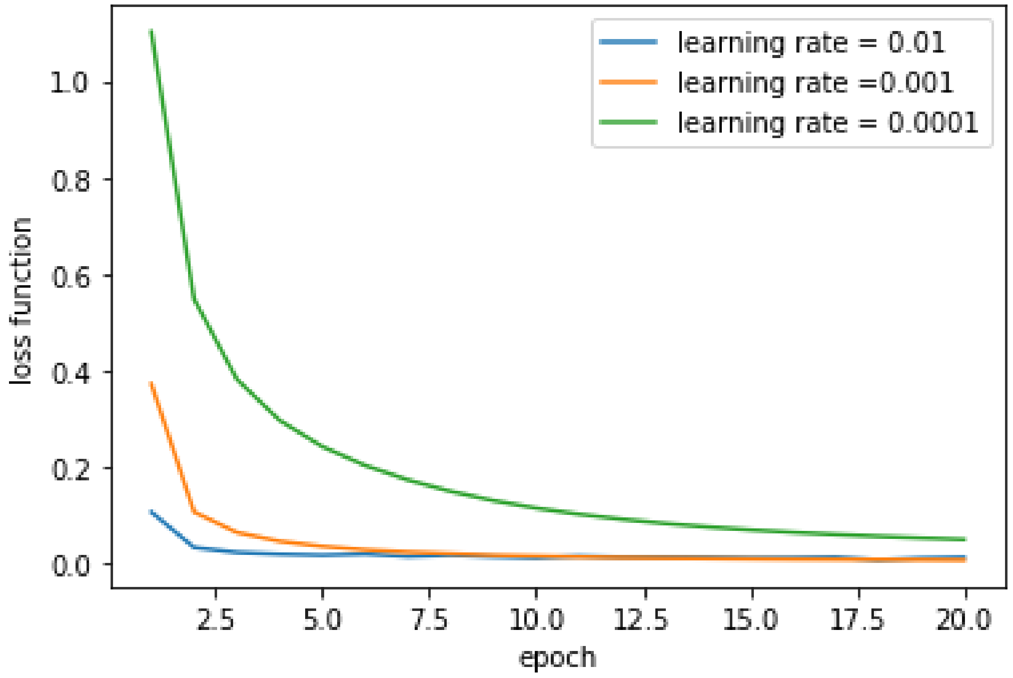
```
663/663 [==============================] - 2s 3ms/step - loss: 0.2952 -
accuracy: 0.9342 - val_loss: 0.3203 - val_accuracy: 0.9259
Epoch 5/20
663/663 [==============================] - 2s 3ms/step - loss: 0.2405 -
accuracy: 0.9461 - val_loss: 0.2668 - val_accuracy: 0.9399
Epoch 6/20
663/663 [==============================] - 2s 3ms/step - loss: 0.2014 -
accuracy: 0.9540 - val_loss: 0.2296 - val_accuracy: 0.9502
Epoch 7/20
663/663 [==============================] - 2s 3ms/step - loss: 0.1714 -
accuracy: 0.9604 - val_loss: 0.2024 - val_accuracy: 0.9521
Epoch 8/20
663/663 [==============================] - 2s 3ms/step - loss: 0.1476 -
accuracy: 0.9654 - val_loss: 0.1823 - val_accuracy: 0.9544
Epoch 9/20
663/663 [==============================] - 2s 3ms/step - loss: 0.1285 -
accuracy: 0.9695 - val_loss: 0.1648 - val_accuracy: 0.9565
Epoch 10/20
663/663 [==============================] - 2s 3ms/step - loss: 0.1129 -
accuracy: 0.9736 - val_loss: 0.1577 - val_accuracy: 0.9545
Epoch 11/20
663/663 [==============================] - 2s 3ms/step - loss: 0.1002 -
accuracy: 0.9763 - val_loss: 0.1439 - val_accuracy: 0.9559
Epoch 12/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0898 -
accuracy: 0.9789 - val_loss: 0.1354 - val_accuracy: 0.9576
Epoch 13/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0810 -
accuracy: 0.9810 - val_loss: 0.1380 - val_accuracy: 0.9547
Epoch 14/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0738 -
accuracy: 0.9828 - val_loss: 0.1356 - val_accuracy: 0.9550
Epoch 15/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0676 -
accuracy: 0.9842 - val_loss: 0.1315 - val_accuracy: 0.9552
Epoch 16/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0625 -
accuracy: 0.9855 - val_loss: 0.1284 - val_accuracy: 0.9558
Epoch 17/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0580 -
accuracy: 0.9863 - val_loss: 0.1250 - val_accuracy: 0.9560
Epoch 18/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0541 -
accuracy: 0.9871 - val_loss: 0.1233 - val_accuracy: 0.9563
Epoch 19/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0507 -
accuracy: 0.9878 - val_loss: 0.1210 - val_accuracy: 0.9579
Epoch 20/20
```

```
663/663 [==============================] - 2s 3ms/step - loss: 0.0477 -
accuracy: 0.9885 - val_loss: 0.1224 - val_accuracy: 0.9562
```

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.

[89]:
```
# TODO 13
epochs=np.arange(start=1, stop=21)
plt.plot(epochs,loss_hist[0])
plt.plot(epochs,loss_hist[1])
plt.plot(epochs,loss_hist[2])
plt.ylabel('loss function')
plt.xlabel('epoch')
plt.legend(['learning rate = '+str(rates[0]), 'learning rate␣
 ↪='+str(rates[1]),'learning rate = '+str(rates[2])])
```

[89]:
```
<matplotlib.legend.Legend at 0x7f787e6843d0>
```



[ ]:
```
```

[ ]:
```
```