Lab: Neural Networks for Music Classification

In addition to the concepts in the MNIST neural network demo (./demo2_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 Tensorflow

Before starting this lab, you will need to install Tensorflow (https://www.tensorflow.org/install/). If you are using Google colaboratory (https://colab.research.google.com), Tensorflow is already installed. Run the following command to ensure Tensorflow is installed.

In [1]:

```
import torch
```

Then, load the other packages.

In [2]:

```
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 [3]:

```
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 [4]:

```
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 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` .

In [5]:

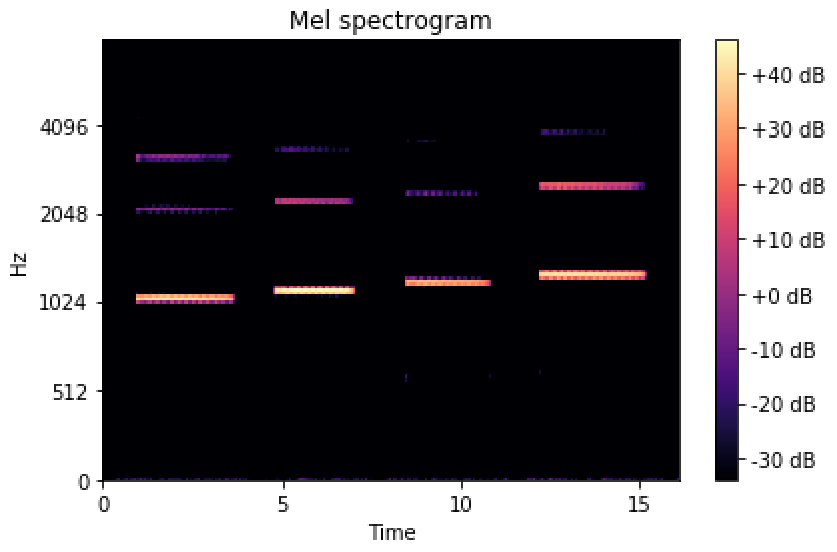
```
# TODO
# y, sr = ...
y, sr = librosa.load(fn)
```

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 [6]:

```
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 [7]:

```
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 [8]:

```
# TODO
print(
    f"No. of training data is {Xtr.shape[0]}, no. of test data is {Xts.shape[0]}."
)
print(f"Each sample contains {Xtr.shape[1]} features.")
print(f"There are {np.unique(ytr).shape[0]} classes per class.")
```

```
No. of training data is 66247, no. of test data is 14904.
Each sample contains 120 features.
There are 10 classes 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.

In [9]:

```
# TODO Scale the training and test matrices
# Xtr_scale = ...
# Xts_scale = ...
mean = np.mean(Xtr, axis=0)
std = np.std(Xtr, axis=0)
Xtr_scale = (Xtr - mean) / std
Xts_scale = (Xts - mean) / std
```

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 [10]:

```
from torch import nn, optim
import torch.utils.data as utils
```

In [33]:

```
# TODO: construct the model
nh = 256
features = Xtr.shape[1]
classes = np.unique(ytr).shape[0]
model = nn.Sequential(nn.Linear(features, nh), nn.Linear(nh, classes))
model = model.double()
```

In [34]:

```
# TODO:  Print the model summary
print(model)
```

```
Sequential(
  (0): Linear(in_features=120, out_features=256, bias=True)
  (1): Linear(in_features=256, out_features=10, bias=True)
)
```

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 [35]:

```
# TODO
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()
```

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 [37]:

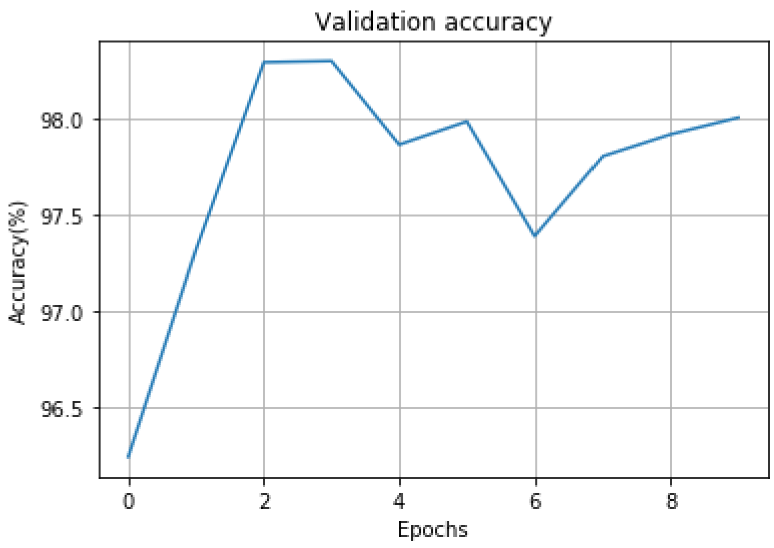
```python
# TODO
epoches = 10

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Running on:", device)

Xtr_scalet = torch.from_numpy(Xtr_scale)
ytrt = torch.from_numpy(ytr)
Xts_scalet = torch.from_numpy(Xts_scale)
ytst = torch.from_numpy(yts)
# convert numpy to tensor
trainset = utils.TensorDataset(Xtr_scalet, ytrt)
testset = utils.TensorDataset(Xts_scalet, ytst)

val_acc = []
losses = []

for epoch in range(epoches):

    trainloader = utils.DataLoader(trainset, batch_size=100, shuffle=True)
    testloader = utils.DataLoader(testset, batch_size=100)

    model.to(device)
    running_loss = 0
    validation_loss = 0
    for i, data in enumerate(trainloader, 0):
        model.train()
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        outputs = model(inputs)

        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()

        if 0 and i % 2000 == 0:  # print every 2000 mini-batches
            print(
                f"Training on {{epoch {epoch + 1}, batchs {i}}}, loss is {running_loss / i *100:.2
            )

    # envaluation

    model.eval()
    train_correct = 0
    train_total = 0
    val_correct = 0
    val_total = 0

    with torch.no_grad():
        for data in trainloader:
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            train_total += labels.size(0)
            train_correct += (predicted == labels).sum().item()

    with torch.no_grad():
        for data in testloader:
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            val_total += labels.size(0)
            val_correct += (predicted == labels).sum().item()
    val_acc.append(100 * val_correct / val_total)
    losses.append(running_loss / i * 100)
    print(
        f"Epoch {epoch+1}/{epoches}: Accuracy on training data is {100 * train_correct / train_tot
loss on training data is {running_loss / i *100:.2f}%, \
accuracy on test data is {100 * val_correct / val_total:.2f}%.")
```

```
Running on: cuda:0
Epoch 0/10: Accuracy on training data is 97.46%, loss on training data is 9.09%, acc
uracy on test data is 96.24%.
Epoch 1/10: Accuracy on training data is 98.03%, loss on training data is 7.08%, acc
uracy on test data is 97.32%.
Epoch 2/10: Accuracy on training data is 98.76%, loss on training data is 6.07%, acc
uracy on test data is 98.30%.
Epoch 3/10: Accuracy on training data is 98.74%, loss on training data is 5.42%, acc
uracy on test data is 98.30%.
Epoch 4/10: Accuracy on training data is 98.82%, loss on training data is 4.86%, acc
uracy on test data is 97.87%.
Epoch 5/10: Accuracy on training data is 98.66%, loss on training data is 4.55%, acc
uracy on test data is 97.99%.
Epoch 6/10: Accuracy on training data is 98.56%, loss on training data is 4.54%, acc
uracy on test data is 97.39%.
Epoch 7/10: Accuracy on training data is 98.66%, loss on training data is 4.12%, acc
uracy on test data is 97.81%.
Epoch 8/10: Accuracy on training data is 98.90%, loss on training data is 4.10%, acc
uracy on test data is 97.92%.
Epoch 9/10: Accuracy on training data is 98.75%, loss on training data is 3.80%, acc
uracy on test data is 98.01%.
```

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.

In [46]:

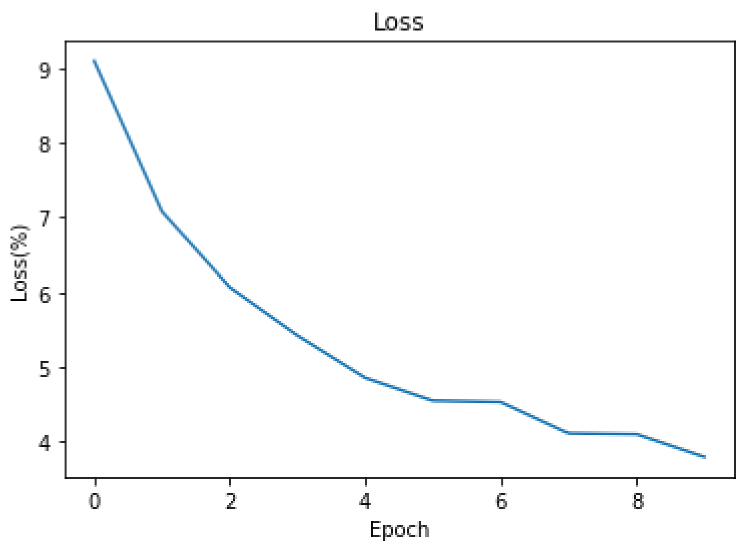
```
# TODO
plt.plot(val_acc)
plt.xlabel("Epochs")
plt.ylabel("Accuracy(%)")
plt.grid()
plt.title("Validation accuracy")
plt.show()
```



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

In [45]:

```
# TODO
plt.plot(losses)
plt.xlabel("Epochs")
plt.ylabel("Loss(%)")
plt.grid()
plt.title("Loss")
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 for 20 epochs
- save the accuracy and losses

In [59]:

```
rates = [0.01, 0.001, 0.0001]
batch_size = 100
loss_hist = []
acc_hist = []

# TODO
# for lr in rate:
#    ...

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Running on:", device)
for lr in rates:
    print(f"Learning rate: {lr}")
    model = nn.Sequential(nn.Linear(features, nh), nn.Linear(nh, classes))
    model = model.double()

    optimizer = optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    epoches = 20

    Xtr_scalet = torch.from_numpy(Xtr_scale)
    ytrt = torch.from_numpy(ytr)
    Xts_scalet = torch.from_numpy(Xts_scale)
    ytst = torch.from_numpy(yts)
    # convert numpy to tensor
    trainset = utils.TensorDataset(Xtr_scalet, ytrt)
    testset = utils.TensorDataset(Xts_scalet, ytst)

    val_acc = []
    losses = []

    for epoch in range(epoches):

        trainloader = utils.DataLoader(trainset, batch_size=100, shuffle=True)
        testloader = utils.DataLoader(testset, batch_size=100)

        model.to(device)
        running_loss = 0
        validation_loss = 0
        for i, data in enumerate(trainloader, 0):
            model.train()
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()

            outputs = model(inputs)

            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()

            if 0 and i % 2000 == 0:  # print every 2000 mini-batches
                print(
                    f"Training on {{epoch {epoch + 1}, batchs {i}}}, loss is {running_loss / i *10
                )

        # envaluation

        model.eval()
        train_correct = 0
        train_total = 0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for data in trainloader:
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)

                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                train_total += labels.size(0)
                train_correct += (predicted == labels).sum().item()

        with torch.no_grad():
            for data in testloader:
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)

                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                val_total += labels.size(0)
                val_correct += (predicted == labels).sum().item()
        val_acc.append(100 * val_correct / val_total)
        losses.append(running_loss / i * 100)
        print(
            f"Epoch {epoch+1}/{epoches}: Accuracy on training data is {100 * train_correct / train
loss on training data is {running_loss / i *100:.2f}%, \
accuracy on test data is {100 * val_correct / val_total:.2f}%.")

    loss_hist.append(losses)
    acc_hist.append(val_acc)
```

```
Running on: cuda:0
Learning rate: 0.01
Epoch 1/20: Accuracy on training data is 94.48%, loss on training data is 26.64%, ac
curacy on test data is 94.52%.
Epoch 2/20: Accuracy on training data is 94.62%, loss on training data is 20.33%, ac
curacy on test data is 91.22%.
Epoch 3/20: Accuracy on training data is 97.53%, loss on training data is 16.37%, ac
curacy on test data is 96.79%.
Epoch 4/20: Accuracy on training data is 96.80%, loss on training data is 16.64%, ac
curacy on test data is 96.37%.
Epoch 5/20: Accuracy on training data is 98.37%, loss on training data is 18.58%, ac
curacy on test data is 97.49%.
Epoch 6/20: Accuracy on training data is 97.17%, loss on training data is 12.21%, ac
curacy on test data is 97.38%.
Epoch 7/20: Accuracy on training data is 97.37%, loss on training data is 21.72%, ac
curacy on test data is 97.28%.
Epoch 8/20: Accuracy on training data is 98.17%, loss on training data is 11.29%, ac
curacy on test data is 97.21%.
Epoch 9/20: Accuracy on training data is 98.34%, loss on training data is 12.06%, ac

curacy on test data is 97.29%.
```

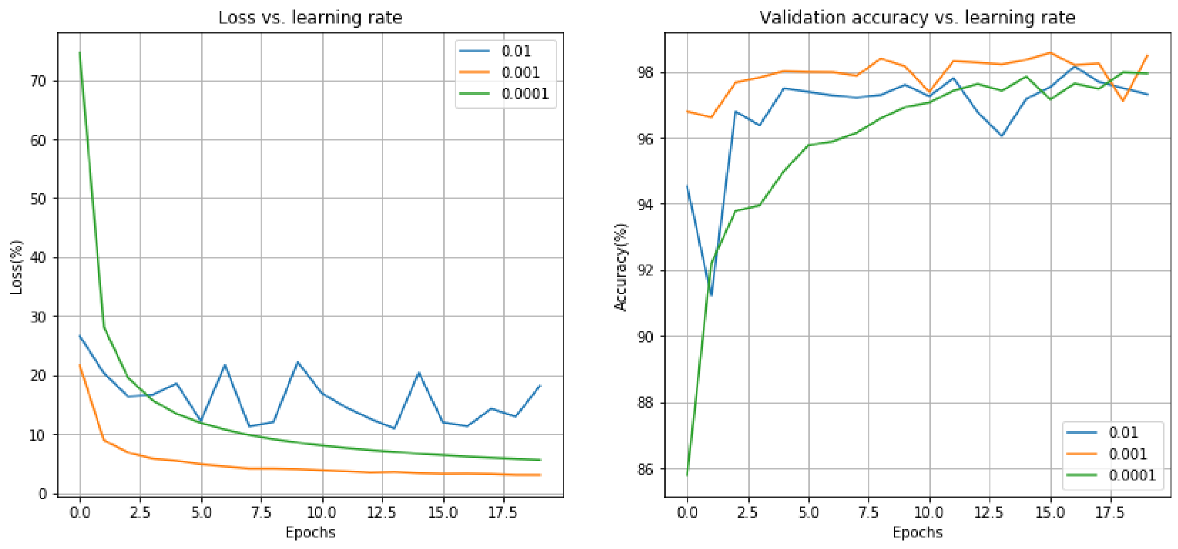
```
Epoch 10/20: Accuracy on training data is 97.59%, loss on training data is 22.23%, a
ccuracy on test data is 97.60%.
Epoch 11/20: Accuracy on training data is 97.82%, loss on training data is 16.92%, a
ccuracy on test data is 97.25%.
Epoch 12/20: Accuracy on training data is 98.12%, loss on training data is 14.52%, a
ccuracy on test data is 97.80%.
Epoch 13/20: Accuracy on training data is 98.41%, loss on training data is 12.59%, a
ccuracy on test data is 96.77%.
Epoch 14/20: Accuracy on training data is 98.34%, loss on training data is 10.95%, a
ccuracy on test data is 96.05%.
Epoch 15/20: Accuracy on training data is 98.14%, loss on training data is 20.41%, a
ccuracy on test data is 97.17%.
Epoch 16/20: Accuracy on training data is 98.29%, loss on training data is 12.00%, a
ccuracy on test data is 97.53%.
Epoch 17/20: Accuracy on training data is 98.71%, loss on training data is 11.33%, a
ccuracy on test data is 98.15%.
Epoch 18/20: Accuracy on training data is 97.95%, loss on training data is 14.31%, a
ccuracy on test data is 97.69%.
Epoch 19/20: Accuracy on training data is 98.85%, loss on training data is 12.95%, a
ccuracy on test data is 97.50%.
Epoch 20/20: Accuracy on training data is 98.52%, loss on training data is 18.20%, a
ccuracy on test data is 97.31%.
Learning rate: 0.001
Epoch 1/20: Accuracy on training data is 97.51%, loss on training data is 21.67%, ac
curacy on test data is 96.79%.
Epoch 2/20: Accuracy on training data is 97.95%, loss on training data is 8.96%, acc
uracy on test data is 96.61%.
Epoch 3/20: Accuracy on training data is 97.98%, loss on training data is 6.88%, acc
uracy on test data is 97.67%.
Epoch 4/20: Accuracy on training data is 98.49%, loss on training data is 5.86%, acc
uracy on test data is 97.82%.
Epoch 5/20: Accuracy on training data is 98.74%, loss on training data is 5.49%, acc
uracy on test data is 98.01%.
Epoch 6/20: Accuracy on training data is 98.63%, loss on training data is 4.89%, acc
uracy on test data is 97.99%.
Epoch 7/20: Accuracy on training data is 98.82%, loss on training data is 4.52%, acc
uracy on test data is 97.98%.
Epoch 8/20: Accuracy on training data is 98.82%, loss on training data is 4.16%, acc
uracy on test data is 97.87%.
Epoch 9/20: Accuracy on training data is 98.82%, loss on training data is 4.16%, acc
uracy on test data is 98.39%.
Epoch 10/20: Accuracy on training data is 98.95%, loss on training data is 4.07%, ac
curacy on test data is 98.15%.
Epoch 11/20: Accuracy on training data is 98.76%, loss on training data is 3.88%, ac
curacy on test data is 97.38%.
Epoch 12/20: Accuracy on training data is 99.10%, loss on training data is 3.71%, ac
curacy on test data is 98.32%.
Epoch 13/20: Accuracy on training data is 98.57%, loss on training data is 3.48%, ac
curacy on test data is 98.27%.
Epoch 14/20: Accuracy on training data is 98.96%, loss on training data is 3.58%, ac
curacy on test data is 98.22%.
Epoch 15/20: Accuracy on training data is 98.81%, loss on training data is 3.38%, ac
curacy on test data is 98.36%.
Epoch 16/20: Accuracy on training data is 99.14%, loss on training data is 3.28%, ac
curacy on test data is 98.56%.
Epoch 17/20: Accuracy on training data is 99.14%, loss on training data is 3.30%, ac
curacy on test data is 98.20%.
Epoch 18/20: Accuracy on training data is 99.08%, loss on training data is 3.23%, ac
curacy on test data is 98.24%.
Epoch 19/20: Accuracy on training data is 98.80%, loss on training data is 3.08%, ac
curacy on test data is 97.10%.
```

```
Epoch 20/20: Accuracy on training data is 99.37%, loss on training data is 3.07%, ac
curacy on test data is 98.48%.
Learning rate: 0.0001
Epoch 1/20: Accuracy on training data is 91.58%, loss on training data is 74.62%, ac
curacy on test data is 85.80%.
Epoch 2/20: Accuracy on training data is 94.71%, loss on training data is 28.13%, ac
curacy on test data is 92.20%.
Epoch 3/20: Accuracy on training data is 95.84%, loss on training data is 19.53%, ac
curacy on test data is 93.78%.
Epoch 4/20: Accuracy on training data is 96.18%, loss on training data is 15.75%, ac
curacy on test data is 93.95%.
Epoch 5/20: Accuracy on training data is 96.65%, loss on training data is 13.44%, ac
curacy on test data is 94.99%.
Epoch 6/20: Accuracy on training data is 97.17%, loss on training data is 11.90%, ac
curacy on test data is 95.77%.
Epoch 7/20: Accuracy on training data is 97.30%, loss on training data is 10.77%, ac
curacy on test data is 95.88%.
Epoch 8/20: Accuracy on training data is 97.53%, loss on training data is 9.83%, acc
uracy on test data is 96.15%.
Epoch 9/20: Accuracy on training data is 97.70%, loss on training data is 9.14%, acc
uracy on test data is 96.58%.
Epoch 10/20: Accuracy on training data is 97.87%, loss on training data is 8.56%, ac
curacy on test data is 96.92%.
Epoch 11/20: Accuracy on training data is 98.05%, loss on training data is 8.07%, ac
curacy on test data is 97.06%.
Epoch 12/20: Accuracy on training data is 98.12%, loss on training data is 7.65%, ac
curacy on test data is 97.42%.
Epoch 13/20: Accuracy on training data is 98.25%, loss on training data is 7.29%, ac
curacy on test data is 97.62%.
Epoch 14/20: Accuracy on training data is 98.26%, loss on training data is 6.97%, ac
curacy on test data is 97.42%.
Epoch 15/20: Accuracy on training data is 98.41%, loss on training data is 6.70%, ac
curacy on test data is 97.85%.
Epoch 16/20: Accuracy on training data is 98.22%, loss on training data is 6.44%, ac
curacy on test data is 97.16%.
Epoch 17/20: Accuracy on training data is 98.39%, loss on training data is 6.19%, ac
curacy on test data is 97.64%.
Epoch 18/20: Accuracy on training data is 98.39%, loss on training data is 6.00%, ac
curacy on test data is 97.48%.
Epoch 19/20: Accuracy on training data is 98.56%, loss on training data is 5.81%, ac
curacy on test data is 97.97%.
Epoch 20/20: Accuracy on training data is 98.57%, loss on training data is 5.64%, ac
curacy on test data is 97.94%.
```

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 [60]:

```python
# TODO
plt.figure(figsize=(14,6))
plt.subplot(1,2,1)
for loss in loss_hist:
    plt.plot(loss)

plt.xlabel("Epochs")
plt.ylabel("Loss(%)")
plt.grid()
plt.title("Loss vs. learning rate")
plt.legend(rates)

plt.subplot(1,2,2)
for acc in acc_hist:
    plt.plot(acc)

plt.xlabel("Epochs")
plt.ylabel("Accuracy(%)")
plt.grid()
plt.title("Validation accuracy vs. learning rate")
plt.legend(rates)
plt.show()
```


Loss vs. learning rate
Loss(%)
Epochs
Validation accuracy vs. learning rate
Accuracy(%)
Epochs