1.EXERCISES IN BACKPROPAGATION

1. Assume we have a n-dimensional input $x = [x_1, x_2, \ldots, x_n]^T$. And we have m units in the hidden layers and the activation function of each unit is sigmoid. And we have q-dimensional output $\hat{y} = [y_1, y_2, \ldots, y_q]^T$.
   The neural network looks like this:

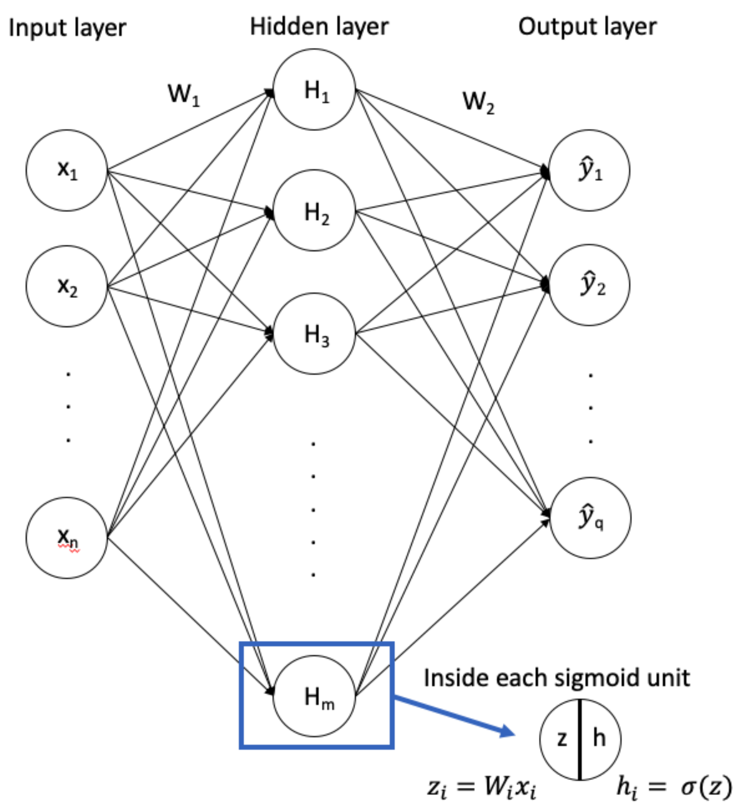


Figure 1. Network Layout

In each unit of hidden layer, there are two values: one is z, the input value of this unit; another one is h, the output value of this unit.
Here is computation graph:

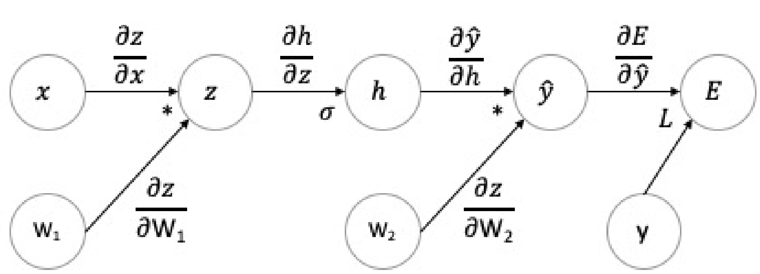


Figure 2. Computation graph

In the forward passes, we can calculate the forward values.

$$z = W_1 x$$

$$h = \sigma(z) = \sigma(W_1 x), \sigma(x) = \frac{1}{1+e^{-x}}$$

$$\hat{y} = W_2 h = W_2 \sigma(w_1 x)$$

$$E = ||\hat{y} - y||_2^2$$

In the backward passes, we need to calculate these partial derivatives first: $\frac{\partial z}{\partial x}$, $\frac{\partial z}{\partial W_1}$, $\frac{\partial h}{\partial z}$, $\frac{\partial \hat{y}}{\partial h}$, $\frac{\partial \hat{y}}{\partial W_2}$, $\frac{\partial E}{\partial \hat{y}}$

Here is how we calculate $\frac{\partial E}{\partial \hat{y}}$:

$$E = \sqrt{(\hat{y_1} - y_1)^2 + (\hat{y_2} - y_2)^2 + \cdots + (\hat{y_q} - y_q)^2}^2 = (\hat{y_1} - y_1)^2 + (\hat{y_2} - y_2)^2 + \cdots + (\hat{y_q} - y_q)^2$$

$$\frac{\partial E}{\partial \hat{y}} = [\frac{\partial E}{\partial \hat{y_1}}, \frac{\partial E}{\partial \hat{y_2}}, \ldots, \frac{\partial E}{\partial \hat{y_q}}]$$

For each partial derivatives,

$$\frac{\partial E}{\partial \hat{y_i}} = 2(\hat{y_i} - y_i)$$

Thus,

$$\frac{\partial E}{\partial \hat{y}} = [2(\hat{y_1} - y_1), 2(\hat{y_2} - y_2), \ldots, 2(\hat{y_q} - y_q)] = 2(\hat{y} - y)$$

Here is how we calculate $\frac{\partial h}{\partial z}$:

$$h = \sigma(z)$$

$$h = \begin{bmatrix} h_1 \\ h_2 \\ \cdots \\ h_i \\ \cdots \\ h_m \end{bmatrix} = \sigma \begin{bmatrix} z_1 \\ z_2 \\ \cdots \\ z_i \\ \cdots \\ z_m \end{bmatrix}$$

We can get the vector Jacobian

$$J = \frac{\partial h}{\partial z} = \begin{bmatrix} \frac{\partial h_1}{\partial z_1} & \frac{\partial h_1}{\partial z_2} & \cdots & \frac{\partial h_1}{\partial z_m} \\ \frac{\partial h_2}{\partial z_1} & \frac{\partial h_2}{\partial z_2} & \cdots & \frac{\partial h_2}{\partial z_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_n}{\partial z_1} & \frac{\partial h_n}{\partial z_2} & \cdots & \frac{\partial h_n}{\partial z_m} \end{bmatrix}$$

Because $h_i = \frac{1}{1+e^{-z_i}}$
Thus, When $i \neq j$, $\frac{\partial h_i}{\partial z_j} = 0$
Thus, J is a diagonal matrix. When $i = j$, $J_{ij} = \frac{\partial h_i}{\partial z_j} = \frac{1}{1+e^{-z_j}}(1 - \frac{1}{1+e^{-z_j}})$

$$J = \frac{\partial h}{\partial z} = \begin{bmatrix} \frac{1}{1+e^{-z_1}}(1 - \frac{1}{1+e^{-z_1}}) & & \\ & \ddots & \\ & & \frac{1}{1+e^{-z_m}}(1 - \frac{1}{1+e^{-z_m}}) \end{bmatrix}$$

Here is how we calculate $\frac{\partial Z}{\partial W_1}$:

$$Z = W^1 x$$

, here we use W^1 to represent W_1

$$z = \begin{bmatrix} z_1 \\ z_2 \\ \dots \\ z_i \\ \dots \\ z_m \end{bmatrix} = \begin{bmatrix} W_{11}^1 & W_{12}^1 & \cdots & W_{1n}^1 \\ W_{21}^1 & W_{22}^1 & \cdots & W_{2n}^1 \\ \vdots & \vdots & \ddots & \vdots \\ W_{m1}^1 & W_{12}^1 & \cdots & W_{mn}^1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}$$

We can get the vector Jacobian

$$J = \frac{\partial z}{\partial W_1} = \begin{bmatrix} \frac{\partial z_1}{\partial W_{1k}^1} & \frac{\partial z_1}{\partial W_{2k}^1} & \cdots & \frac{\partial z_1}{\partial \partial W_{mk}^1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_n}{\partial W_{1k}^1} & \frac{\partial z_n}{\partial W_{2k}^1} & \cdots & \frac{\partial z_n}{\partial W_{mk}^1} \end{bmatrix}$$

And what we need to address is that the size of J is $m*(m*n)$, and

$$\frac{\partial z_i}{\partial W_j^1} = [\frac{\partial z_i}{\partial W_{j1}^1}, \frac{\partial z_i}{\partial W_{j2}^1}, \dots, \frac{\partial z_i}{\partial W_{jn}^1}]$$

Because $z_i = W_{i1}^1 x_1 + W_{i2}^1 x_2 + \cdots + W_{in}^1 x_n$
Thus, When $i \neq j$, $\frac{\partial z_i}{\partial W_{jk}^1} = 0$
When $i = j$, $\frac{\partial z_i}{\partial W_{jk}^1} = x_k, k = 1, 2, \dots, n$

$$\frac{\partial z_i}{\partial W_j^1} = [x_1, x_2, ..x_n] = x^T$$

Thus, J is a diagonal matrix. When $i = j$, $J_{ij} = x^T$

$$J = \frac{\partial z}{\partial W_1} = \begin{bmatrix} [x_1, \dots, x_n] & & \\ & \ddots & \\ & & [x_1, \dots, x_n] \end{bmatrix}$$

We can get $\frac{\partial \hat{y}}{\partial W_2}$ in the same way.

$$\frac{\partial \hat{y}}{\partial W_2} = \begin{bmatrix} [h_1, \dots, h_m] & & \\ & \ddots & \\ & & [h_1, \dots, h_m] \end{bmatrix}$$

Here is how we calculate $\frac{\partial Z}{\partial x}$:
We can get the vector Jacobian

$$J = \frac{\partial z}{\partial x} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \frac{\partial z_1}{\partial x_2} & \cdots & \frac{\partial z_1}{\partial \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial z_n}{\partial x_1} & \frac{\partial z_n}{\partial x_2} & \cdots & \frac{\partial z_n}{\partial x_n} \end{bmatrix}$$

Because $z_i = W_{i1}^1 x_1 + W_{i2}^1 x_2 + \cdots + W_{in}^1 x_n$
Thus,

$$\frac{\partial z_i}{\partial x_j} = W_{ij}^1$$

Thus,

$$J = \frac{\partial z}{\partial x} = W_1$$

We can get $\frac{\partial \hat{y}}{\partial h}$ in the same way.

$$\frac{\partial \hat{y}}{\partial h} = W_2$$

Finally, Our goal is to update W_1 and W_2, so we need to get $\frac{\partial E}{\partial W_1}$ and $\frac{\partial E}{\partial W_2}$.

$$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial W_2}$$

$$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial h} * \frac{\partial h}{\partial z} * \frac{\partial z}{\partial W_1}$$

And other derivatives we can get are:

$$\frac{\partial E}{\partial \hat{y}}$$

$$\frac{\partial E}{\partial h} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial h}$$

$$\frac{\partial E}{\partial z} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial h} * \frac{\partial h}{\partial z}$$

$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial h} * \frac{\partial h}{\partial z} * \frac{\partial z}{\partial x}$$

And all the partial derivatives were calculated above. We can multiply them one by one to get these results.
As we can see, the most expensive pass is to get $\frac{\partial E}{\partial W_1}$. Let's compare $\frac{\partial E}{\partial W_1}$ to $\frac{\partial E}{\partial W_2}$, since the computation of $\frac{\partial z}{\partial W_1}$ and $\frac{\partial \hat{y}}{\partial W_2}$ is very similar. Thus computing $\frac{\partial E}{\partial W_1}$ has two more operations($\frac{\partial \hat{y}}{\partial h}$ and $\frac{\partial h}{\partial z}$) than computing $\frac{\partial E}{\partial W_2}$. And the computing complexity of $\frac{\partial E}{\partial \hat{y}}$ and $\frac{\partial \hat{y}}{\partial h}$ are similar and the computing complexity of $\frac{\partial h}{\partial z}$, $\frac{\partial z}{\partial w_1}$ and $\frac{\partial \hat{y}}{\partial w_2}$ are similar .Thus, computing $\frac{\partial E}{\partial W_1}$ is about 2 times more expensive than computing $\frac{\partial E}{\partial W_2}$.

2.SLOW RATE OF DESCENT

1. Here is the gradient.

$$L(w_1, w_2) = 0.5(aw_1^2 + bw_2^2)$$

$$\nabla L(w_1, w_2) = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}]$$

$$\frac{\partial L}{\partial w_1} = 2 * 0.5 * aw_1 = aw_1$$

$$\frac{\partial L}{\partial w_2} = 2 * 0.5 * bw_2 = bw_2$$

$$\nabla L(w_1, w_2) = [aw_1, bw_2]$$

We can assume that a and b are positive.
When $\nabla L(w_1, w_2) = [aw_1, bw_2] = 0$, which means that $w_1 = w_2 = 0$, then we can achieve the minimum value of L.

2.

$$w_1(t+1) = w_1(t) - \eta \nabla L(w_1(t))$$

$$w_1(t+1) = w_1(t) - \eta a w_1(t) = (1 - a\eta) w_1(t) = \rho_1 w_1(t)$$

$$\rho_1 = 1 - a\eta$$

In the same way, We can get $\rho_1 = 1 - b\eta$

3. If we want the gradient descent converge, we need to meet this requirement:

$$\lim_{t\to\infty} = \frac{w_1(t+1) - w_1^*}{w_1(t) - w^*} = \frac{(1-a\eta)w_1(t) - w_1^*}{w_1(t) - w_1^*} = \mu, 0 < \mu < 1$$

Thus,

$$1 - a\eta \to \mu$$

Then

$$0 < 1 - a\eta < 1$$

$$0 < \eta < \frac{1}{a}$$

In the same way we can get that

$$0 < \eta < \frac{1}{b}$$

Thus,

$$0 < \eta < min(\frac{1}{a}, \frac{1}{b})$$

4. If $\frac{a}{b}$ is a very large ratio, for example, if $\frac{a}{b} = h, h \to \infty$ Then

$$a = hb$$

Then

$$0 < \eta < min(\frac{1}{hb}, \frac{1}{b}) = min\frac{1}{hb}$$

$$\because \frac{1}{hb} \to 0, \therefore \eta \to 0$$

Thus, the learning rate is very small and the convergence rate of gradient descent is very slow.

Neural Networks for Musical Instrument Classification

In this assignment, we will attempt a musical instrument classification problem. Given a sample of music, we want to determine which instrument (e.g. trumpet, violin, piano) is playing.

This assignment is closely based on one by Sundeep Rangan, from his [IntroML GitHub repo (https://github.com/sdrangan/introml/)](https://github.com/sdrangan/introml/).

In [2]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Audio Feature Extraction with Librosa

The key to audio classification is to extract useful features. The `librosa` package in Python has a rich set of methods for extracting the features of audio samples commonly used in machine learning tasks, such as speech recognition and sound classification.

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)](http://theremin.music.uiowa.edu)

We will use the `wget` command to retrieve one file to our Google Colab storage area. (We can run `wget` and many other basic Linux commands in Colab by prefixing them with a `!` or `%` .)

In [4]:
```
!wget "http://theremin.music.uiowa.edu/sound files/MIS/Woodwinds/sopranosaxophone/SopSax.Vib.pp.C6Eb6.aiff"
```

```
--2020-10-02 15:57:57--  http://theremin.music.uiowa.edu/sound%20files/MIS/Woodwinds/sopranosaxophon
e/SopSax.Vib.pp.C6Eb6.aiff
Resolving theremin.music.uiowa.edu (theremin.music.uiowa.edu)... 128.255.102.154, 2620:0:e50:680c::7
3
Connecting to theremin.music.uiowa.edu (theremin.music.uiowa.edu)|128.255.102.154|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1418242 (1.4M) [audio/aiff]
Saving to: ‘SopSax.Vib.pp.C6Eb6.aiff.1’

SopSax.Vib.pp.C6Eb6 100%[===================>]   1.35M  5.73MB/s    in 0.2s

2020-10-02 15:57:57 (5.73 MB/s) - ‘SopSax.Vib.pp.C6Eb6.aiff.1’ saved [1418242/1418242]
```

Now, if you click on the small folder icon on the far left of the Colab interface, you can see the files in your Colab storage. You should see the "SopSax.Vib.pp.C6Eb6.aiff" file appear there.

In order to listen to this file, we'll first convert it into the `wav` format. Again, we'll use the `!` to run a basic command-line utility: `ffmpeg` , a powerful tool for working with audio and video files.

In [5]:
```
aiff_file = 'SopSax.Vib.pp.C6Eb6.aiff'
wav_file = 'SopSax.Vib.pp.C6Eb6.wav'

!ffmpeg -y -i $aiff_file $wav_file
```

```
ffmpeg version 3.4.8-0ubuntu0.2 Copyright (c) 2000-2020 the FFmpeg developers
  built with gcc 7 (Ubuntu 7.5.0-3ubuntu1~18.04)
  configuration: --prefix=/usr --extra-version=0ubuntu0.2 --toolchain=hardened --libdir=/usr/lib/x86
_64-linux-gnu --incdir=/usr/include/x86_64-linux-gnu --enable-gpl --disable-stripping --enable-avres
ample --enable-avisynth --enable-gnutls --enable-ladspa --enable-libass --enable-libbluray --enable-
libbs2b --enable-libcaca --enable-libcdio --enable-libflite --enable-libfontconfig --enable-libfreet
ype --enable-libfribidi --enable-libgme --enable-libgsm --enable-libmp3lame --enable-libmysofa --ena
ble-libopenjpeg --enable-libopenmpt --enable-libopus --enable-libpulse --enable-librubberband --enab
le-librsvg --enable-libshine --enable-libsnappy --enable-libsoxr --enable-libspeex --enable-libssh -
-enable-libtheora --enable-libtwolame --enable-libvorbis --enable-libvpx --enable-libwavpack --enabl
e-libwebp --enable-libx265 --enable-libxml2 --enable-libxvid --enable-libzmq --enable-libzvbi --enab
le-omx --enable-openal --enable-opengl --enable-sdl2 --enable-libdc1394 --enable-libdrm --enable-lib
iec61883 --enable-chromaprint --enable-frei0r --enable-libopencv --enable-libx264 --enable-shared
  libavutil      55. 78.100 / 55. 78.100
  libavcodec     57.107.100 / 57.107.100
  libavformat    57. 83.100 / 57. 83.100
  libavdevice    57. 10.100 / 57. 10.100
  libavfilter     6.107.100 /  6.107.100
  libavresample   3.  7.  0 /  3.  7.  0
  libswscale      4.  8.100 /  4.  8.100
  libswresample   2.  9.100 /  2.  9.100
  libpostproc    54.  7.100 / 54.  7.100
Guessed Channel Layout for Input Stream #0.0 : mono
Input #0, aiff, from 'SopSax.Vib.pp.C6Eb6.aiff':
  Duration: 00:00:16.07, start: 0.000000, bitrate: 705 kb/s
    Stream #0:0: Audio: pcm_s16be, 44100 Hz, mono, s16, 705 kb/s
Stream mapping:
  Stream #0:0 -> #0:0 (pcm_s16be (native) -> pcm_s16le (native))
Press [q] to stop, [?] for help
Output #0, wav, to 'SopSax.Vib.pp.C6Eb6.wav':
  Metadata:
    ISFT            : Lavf57.83.100
    Stream #0:0: Audio: pcm_s16le ([1][0][0][0] / 0x0001), 44100 Hz, mono, s16, 705 kb/s
    Metadata:
      encoder         : Lavc57.107.100 pcm_s16le
size=    1385kB time=00:00:16.07 bitrate= 705.6kbits/s speed=3.37e+03x
video:0kB audio:1384kB subtitle:0kB other streams:0kB global headers:0kB muxing overhead: 0.005502%
```

Now, we can play the file directly from Colab. If you listen to it you will hear a soprano saxaphone (with vibrato) playing four notes (C, C#, D, Eb).

In [6]:
```
import IPython.display as ipd
ipd.Audio(wav_file)
```

Out[6]:

0:00 / 0:16

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

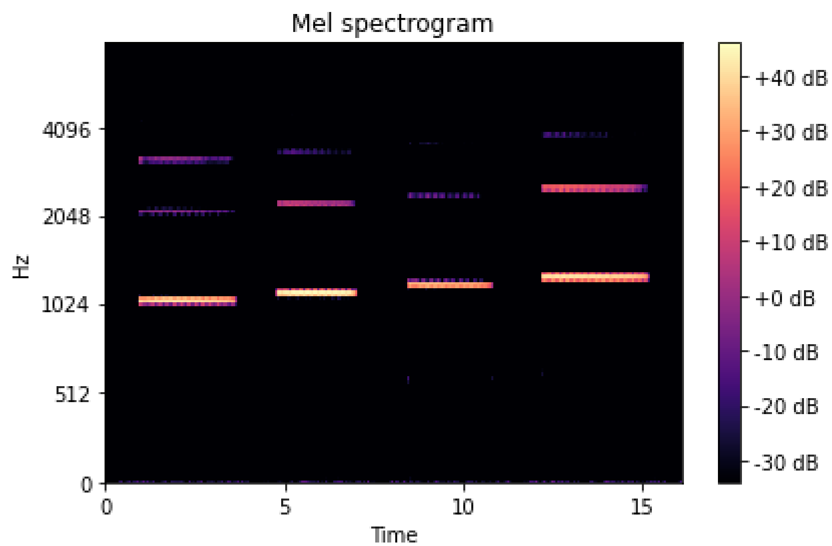
In [7]:
```
y, sr = librosa.load(aiff_file)
```

Feature engineering from audio files is an entire course 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 extracts features that correlate with human audio perception.

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 [8]:
```
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, [Prof. Juan Bello (http://steinhardt.nyu.edu/faculty/Juan_Pablo_Bello)](http://steinhardt.nyu.edu/faculty/Juan_Pablo_Bello) and his former PhD student Eric Humphrey 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.

To retrieve their data, visit

https://github.com/marl/dl4mir-tutorial/blob/master/README.md (https://github.com/marl/dl4mir-tutorial/blob/master/README.md)

and note the password listed on that page. Click on the link for "Instrument Dataset", enter the password, click on `instrument_dataset` to open the folder, and download the four files there. and note the password listed on that page. Click on the link for "Instrument Dataset", enter the password, click on `instrument_dataset` to open the folder, and download the four files there. (You can "direct download" straight from this site, you don't need a Dropbox account.)

Then, upload the files to your Google Colab storage: click on the folder icon on the left to see your storage, if it isn't already open, and then click on "Upload". Wait until *all* uploads have completed.

Then, load the files with:

In [9]:
```
Xtr = np.load('uiowa_train_data.npy')
ytr = np.load('uiowa_train_labels.npy')
Xts = np.load('uiowa_test_data.npy')
yts = np.load('uiowa_test_labels.npy')
```

Examine the data you have just loaded in:

- 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?

Write some code to find these values and print them.

In [61]:
```
# TODO 1
print("The number of training data samples: ", len(Xtr))
print("The number of testing data samples: ",len(Xts))
print("The number of feature of each sample: ",len(Xts[0]))
print("The number of classes: ",len(set(yts)))
```

```
The number of training data samples:  66247
The number of testing data samples:  14904
The number of feature of each sample:  120
The number of classes:  10
```

Then, standardize the training and test data, `Xtr` and `Xts`, by removing the mean of each feature and scaling to unit variance.

You can do this manually, or using `sklearn` 's [StandardScaler (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html)](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html).

Make sure you standardize both the training and test data using the mean and variance of the *training data only*. (If using a `StandardScaler` : create a single `StandardScaler` , call `fit` with the training data, then call `tranform` with the training data, and finally call `transform` with the test data.)

Standardizing input data can make the gradient descent easier; see [this video](https://www.youtube.com/watch?reload=9&v=UIp2CMI0748) for further explanation.

In [39]:
```
# TODO 2 Scale the training and test matrices
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(Xtr)
Xtr_scale  = scaler.transform(Xtr)
Xts_scale  = scaler.transform(Xts)
```

Building a Neural Network Classifier

Following the example in the [demo you have seen (https://colab.research.google.com/drive/1t2OeBGcfB5HSDFl6FPQFaQKbmeEAPPgG?usp=sharing)](https://colab.research.google.com/drive/1t2OeBGcfB5HSDFl6FPQFaQKbmeEAPPgG?usp=sharing), prepare and create a neural network with the following configuration:

- 256 hidden units in a single dense hidden layer
- sigmoid activation at hidden units
- `softmax` activation at the output (since this is a multi-class classification problem)
- Cross-entropy loss
- Adam optimizer with a learning rate of 0.001
- print the model summary

In [55]:
```
# TODO 3 construct the model, print model summary, and compile the model
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Activation
import tensorflow.keras.backend as K
from tensorflow.keras import optimizers
from keras.utils import np_utils

# change the label to one-hot mode
y_train = np_utils.to_categorical(ytr, num_classes=10)
y_test = np_utils.to_categorical(yts, num_classes=10)

K.clear_session()
nin = 120  # dimension of input data
nh = 256  # number of hidden units
nout = 10  # number of outputs = 10 since this is a multi-classification problem and we have 10 class
es
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=0.001)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
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
_________________________________________________________________
```

Fit the model for 10 epochs (passes through the entire data). Use the scaled training data to fit the model, and also pass the test data as "validation data" so that the loss and accuracy will be computed on the test data as well.

Use a batch size of 128. Your final accuracy should be >99%.

In [56]:
```
# TODO 4 fit the model
hist = model.fit(Xtr_scale, y_train, epochs=10, batch_size=128, validation_data=(Xts_scale,y_test))
```

```
Epoch 1/10
518/518 [==============================] - 2s 3ms/step - loss: 0.4057 - accuracy: 0.8878 - val_loss:
0.2293 - val_accuracy: 0.9332
Epoch 2/10
518/518 [==============================] - 1s 3ms/step - loss: 0.1219 - accuracy: 0.9709 - val_loss:
0.1211 - val_accuracy: 0.9622
Epoch 3/10
518/518 [==============================] - 1s 3ms/step - loss: 0.0723 - accuracy: 0.9834 - val_loss:
0.0700 - val_accuracy: 0.9832
Epoch 4/10
518/518 [==============================] - 1s 3ms/step - loss: 0.0508 - accuracy: 0.9878 - val_loss:
0.0565 - val_accuracy: 0.9861
Epoch 5/10
518/518 [==============================] - 1s 3ms/step - loss: 0.0385 - accuracy: 0.9904 - val_loss:
0.0457 - val_accuracy: 0.9883
Epoch 6/10
518/518 [==============================] - 1s 3ms/step - loss: 0.0307 - accuracy: 0.9922 - val_loss:
0.0420 - val_accuracy: 0.9892
Epoch 7/10
518/518 [==============================] - 1s 3ms/step - loss: 0.0251 - accuracy: 0.9935 - val_loss:
0.0359 - val_accuracy: 0.9903
Epoch 8/10
518/518 [==============================] - 1s 3ms/step - loss: 0.0211 - accuracy: 0.9946 - val_loss:
0.0322 - val_accuracy: 0.9903
Epoch 9/10
518/518 [==============================] - 1s 3ms/step - loss: 0.0181 - accuracy: 0.9953 - val_loss:
0.0280 - val_accuracy: 0.9921
Epoch 10/10
518/518 [==============================] - 1s 3ms/step - loss: 0.0156 - accuracy: 0.9957 - val_loss:
0.0275 - val_accuracy: 0.9915
```

Plot the training and test accuracy vs. epochs on one subplot, and the training and test loss vs. epoch on another subplot. Use a log scale for the vertical axis on the loss plot.

You should see that the test accuracy saturates at a little higher than 99%. After that it may "bounce around" due to the noise in the stochastic mini-batch gradient descent.

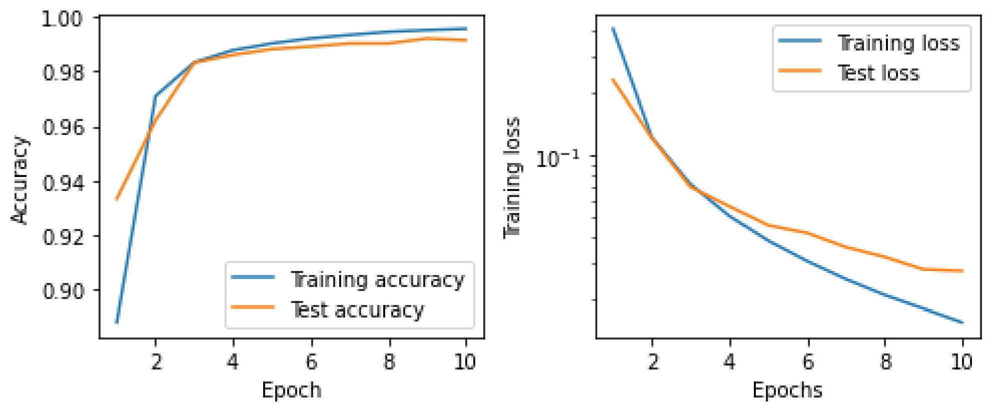
In [57]:
```
# TODO 5 two subplots: one of accuracy vs. epochs, one of loss vs. epochs
# in each subplot, show training in one color and test in another color
import seaborn as sns

plt.figure(figsize=(7,3))

plt.subplot(1,2,1)
train_acc = hist.history['accuracy'];
test_acc = hist.history['val_accuracy'];

nepochs = len(train_acc);
sns.lineplot(x=np.arange(1,nepochs+1), y=train_acc, label='Training accuracy');
sns.lineplot(x=np.arange(1,nepochs+1), y=test_acc, label='Test accuracy');
plt.xlabel('Epoch');
plt.ylabel('Accuracy');

plt.subplot(1,2,2)
train_loss = hist.history['loss']
test_loss = hist.history['val_loss']
sns.lineplot(x=np.arange(1,nepochs+1), y=train_loss, label='Training loss');
sns.lineplot(x=np.arange(1,nepochs+1), y=test_loss, label='Test loss');
plt.yscale('log')
plt.xlabel('Epochs')
plt.ylabel('Training loss')
plt.tight_layout()
```



Varying 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
- prepare a neural network model as described above, with the appropriate learning rate
- train the model for 20 epochs
- save the accuracy and losses

In [58]:
```
rates = [0.1, 0.01,0.001,0.0001]

# TODO 6
train_acc = []
test_acc = []
train_loss = []
test_loss = []

def set_up_model(learning_rate):
  K.clear_session()
  nin = 120  # dimension of input data
  nh = 256  # number of hidden units
  nout = 10  # number of outputs = the number of classes
  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(learning_rate)
  model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])

for lr in rates:
        # set up the model with appropriate learning rate
        set_up_model(lr)
        # train and test
        hist = model.fit(Xtr_scale, y_train, epochs=20, batch_size=128, validation_data=(Xts_scale,y_
test))
        # save the accuracy and losses
        train_acc.append(hist.history['accuracy'])
        test_acc.append(hist.history['val_accuracy'])
        train_loss.append(hist.history['loss'])
        test_loss.append(hist.history['val_loss'])
```

```
Epoch 1/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0140 - accuracy: 0.9963 - val_loss:
0.0292 - val_accuracy: 0.9905
Epoch 2/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0120 - accuracy: 0.9972 - val_loss:
0.0258 - val_accuracy: 0.9906
Epoch 3/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0109 - accuracy: 0.9973 - val_loss:
0.0245 - val_accuracy: 0.9918
Epoch 4/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0098 - accuracy: 0.9978 - val_loss:
0.0210 - val_accuracy: 0.9926
Epoch 5/20
518/518 [==============================] - 2s 3ms/step - loss: 0.0089 - accuracy: 0.9975 - val_loss:
0.0202 - val_accuracy: 0.9925
Epoch 6/20
518/518 [==============================] - 2s 3ms/step - loss: 0.0082 - accuracy: 0.9980 - val_loss:
0.0242 - val_accuracy: 0.9913
Epoch 7/20
518/518 [==============================] - 2s 3ms/step - loss: 0.0077 - accuracy: 0.9980 - val_loss:
0.0210 - val_accuracy: 0.9920
Epoch 8/20
518/518 [==============================] - 2s 3ms/step - loss: 0.0069 - accuracy: 0.9982 - val_loss:
0.0230 - val_accuracy: 0.9917
Epoch 9/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0066 - accuracy: 0.9982 - val_loss:
0.0257 - val_accuracy: 0.9901
Epoch 10/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0059 - accuracy: 0.9985 - val_loss:
0.0231 - val_accuracy: 0.9915
Epoch 11/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0056 - accuracy: 0.9986 - val_loss:
0.0211 - val_accuracy: 0.9925
Epoch 12/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0051 - accuracy: 0.9987 - val_loss:
0.0220 - val_accuracy: 0.9911
Epoch 13/20
518/518 [==============================] - 2s 3ms/step - loss: 0.0048 - accuracy: 0.9988 - val_loss:
0.0225 - val_accuracy: 0.9913
Epoch 14/20
518/518 [==============================] - 2s 3ms/step - loss: 0.0047 - accuracy: 0.9987 - val_loss:
0.0264 - val_accuracy: 0.9909
Epoch 15/20
```

```
518/518 [==============================] - 1s 3ms/step - loss: 0.0042 - accuracy: 0.9990 - val_loss:
0.0234 - val_accuracy: 0.9914
Epoch 16/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0040 - accuracy: 0.9989 - val_loss:
0.0223 - val_accuracy: 0.9919
Epoch 17/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0037 - accuracy: 0.9993 - val_loss:
0.0231 - val_accuracy: 0.9917
Epoch 18/20
518/518 [==============================] - 2s 3ms/step - loss: 0.0037 - accuracy: 0.9990 - val_loss:
0.0190 - val_accuracy: 0.9936
Epoch 19/20
518/518 [==============================] - 2s 4ms/step - loss: 0.0034 - accuracy: 0.9991 - val_loss:
0.0216 - val_accuracy: 0.9928
Epoch 20/20
518/518 [==============================] - 2s 3ms/step - loss: 0.0031 - accuracy: 0.9992 - val_loss:
0.0240 - val_accuracy: 0.9922
Epoch 1/20
518/518 [==============================] - 2s 3ms/step - loss: 0.0027 - accuracy: 0.9994 - val_loss:
0.0218 - val_accuracy: 0.9926
Epoch 2/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0033 - accuracy: 0.9991 - val_loss:
0.0248 - val_accuracy: 0.9917
Epoch 3/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0033 - accuracy: 0.9990 - val_loss:
0.0255 - val_accuracy: 0.9928
Epoch 4/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0027 - accuracy: 0.9992 - val_loss:
0.0268 - val_accuracy: 0.9917
Epoch 5/20
518/518 [==============================] - 1s 2ms/step - loss: 0.0026 - accuracy: 0.9993 - val_loss:
0.0315 - val_accuracy: 0.9895
Epoch 6/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0028 - accuracy: 0.9993 - val_loss:
0.0268 - val_accuracy: 0.9925
Epoch 7/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0021 - accuracy: 0.9997 - val_loss:
0.0249 - val_accuracy: 0.9926
Epoch 8/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0021 - accuracy: 0.9994 - val_loss:
0.0275 - val_accuracy: 0.9928
Epoch 9/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0021 - accuracy: 0.9995 - val_loss:
```

```
0.0275 - val_accuracy: 0.9923
Epoch 10/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0020 - accuracy: 0.9995 - val_loss:
0.0337 - val_accuracy: 0.9897
Epoch 11/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0018 - accuracy: 0.9996 - val_loss:
0.0285 - val_accuracy: 0.9919
Epoch 12/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0018 - accuracy: 0.9996 - val_loss:
0.0413 - val_accuracy: 0.9883
Epoch 13/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0018 - accuracy: 0.9996 - val_loss:
0.0333 - val_accuracy: 0.9918
Epoch 14/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0018 - accuracy: 0.9995 - val_loss:
0.0287 - val_accuracy: 0.9931
Epoch 15/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0018 - accuracy: 0.9995 - val_loss:
0.0315 - val_accuracy: 0.9918
Epoch 16/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0015 - accuracy: 0.9997 - val_loss:
0.0271 - val_accuracy: 0.9934
Epoch 17/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0016 - accuracy: 0.9995 - val_loss:
0.0322 - val_accuracy: 0.9917
Epoch 18/20
518/518 [==============================] - 1s 2ms/step - loss: 0.0014 - accuracy: 0.9996 - val_loss:
0.0421 - val_accuracy: 0.9917
Epoch 19/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0014 - accuracy: 0.9996 - val_loss:
0.0310 - val_accuracy: 0.9923
Epoch 20/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0013 - accuracy: 0.9996 - val_loss:
0.0365 - val_accuracy: 0.9905
Epoch 1/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0013 - accuracy: 0.9997 - val_loss:
0.0308 - val_accuracy: 0.9922
Epoch 2/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0015 - accuracy: 0.9997 - val_loss:
0.0349 - val_accuracy: 0.9910
Epoch 3/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0013 - accuracy: 0.9996 - val_loss:
0.0345 - val_accuracy: 0.9922
```

```
Epoch 4/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0010 - accuracy: 0.9998 - val_loss:
0.0373 - val_accuracy: 0.9914
Epoch 5/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0013 - accuracy: 0.9996 - val_loss:
0.0457 - val_accuracy: 0.9893
Epoch 6/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0013 - accuracy: 0.9997 - val_loss:
0.0334 - val_accuracy: 0.9928
Epoch 7/20
518/518 [==============================] - 1s 3ms/step - loss: 9.8747e-04 - accuracy: 0.9998 - val_l
oss: 0.0381 - val_accuracy: 0.9914
Epoch 8/20
518/518 [==============================] - 1s 3ms/step - loss: 9.1686e-04 - accuracy: 0.9998 - val_l
oss: 0.0384 - val_accuracy: 0.9909
Epoch 9/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0012 - accuracy: 0.9997 - val_loss:
0.0342 - val_accuracy: 0.9926
Epoch 10/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0011 - accuracy: 0.9997 - val_loss:
0.0387 - val_accuracy: 0.9914
Epoch 11/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0013 - accuracy: 0.9996 - val_loss:
0.0360 - val_accuracy: 0.9926
Epoch 12/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0010 - accuracy: 0.9997 - val_loss:
0.0368 - val_accuracy: 0.9927
Epoch 13/20
518/518 [==============================] - 1s 3ms/step - loss: 7.5337e-04 - accuracy: 0.9998 - val_l
oss: 0.0371 - val_accuracy: 0.9927
Epoch 14/20
518/518 [==============================] - 1s 3ms/step - loss: 7.0687e-04 - accuracy: 0.9998 - val_l
oss: 0.0370 - val_accuracy: 0.9924
Epoch 15/20
518/518 [==============================] - 1s 3ms/step - loss: 9.8708e-04 - accuracy: 0.9997 - val_l
oss: 0.0424 - val_accuracy: 0.9909
Epoch 16/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0011 - accuracy: 0.9996 - val_loss:
0.0401 - val_accuracy: 0.9921
Epoch 17/20
518/518 [==============================] - 1s 3ms/step - loss: 6.9552e-04 - accuracy: 0.9998 - val_l
oss: 0.0400 - val_accuracy: 0.9918
Epoch 18/20
```

```
518/518 [==============================] - 1s 3ms/step - loss: 7.3073e-04 - accuracy: 0.9998 - val_l
oss: 0.0396 - val_accuracy: 0.9924
Epoch 19/20
518/518 [==============================] - 1s 3ms/step - loss: 8.7643e-04 - accuracy: 0.9997 - val_l
oss: 0.0404 - val_accuracy: 0.9915
Epoch 20/20
518/518 [==============================] - 1s 3ms/step - loss: 8.3511e-04 - accuracy: 0.9998 - val_l
oss: 0.0412 - val_accuracy: 0.9924
Epoch 1/20
518/518 [==============================] - 1s 3ms/step - loss: 7.5129e-04 - accuracy: 0.9998 - val_l
oss: 0.0413 - val_accuracy: 0.9913
Epoch 2/20
518/518 [==============================] - 1s 3ms/step - loss: 9.3427e-04 - accuracy: 0.9997 - val_l
oss: 0.0350 - val_accuracy: 0.9925
Epoch 3/20
518/518 [==============================] - 1s 3ms/step - loss: 7.3163e-04 - accuracy: 0.9998 - val_l
oss: 0.0391 - val_accuracy: 0.9932
Epoch 4/20
518/518 [==============================] - 1s 3ms/step - loss: 6.6024e-04 - accuracy: 0.9998 - val_l
oss: 0.0387 - val_accuracy: 0.9926
Epoch 5/20
518/518 [==============================] - 1s 2ms/step - loss: 0.0012 - accuracy: 0.9996 - val_loss:
0.0406 - val_accuracy: 0.9919
Epoch 6/20
518/518 [==============================] - 1s 3ms/step - loss: 6.8106e-04 - accuracy: 0.9998 - val_l
oss: 0.0436 - val_accuracy: 0.9919
Epoch 7/20
518/518 [==============================] - 1s 3ms/step - loss: 7.2183e-04 - accuracy: 0.9998 - val_l
oss: 0.0421 - val_accuracy: 0.9922
Epoch 8/20
518/518 [==============================] - 1s 3ms/step - loss: 5.2230e-04 - accuracy: 0.9999 - val_l
oss: 0.0474 - val_accuracy: 0.9911
Epoch 9/20
518/518 [==============================] - 1s 3ms/step - loss: 8.2705e-04 - accuracy: 0.9997 - val_l
oss: 0.0432 - val_accuracy: 0.9917
Epoch 10/20
518/518 [==============================] - 1s 3ms/step - loss: 4.5257e-04 - accuracy: 0.9999 - val_l
oss: 0.0415 - val_accuracy: 0.9922
Epoch 11/20
518/518 [==============================] - 1s 3ms/step - loss: 4.5349e-04 - accuracy: 0.9999 - val_l
oss: 0.0450 - val_accuracy: 0.9916
Epoch 12/20
518/518 [==============================] - 1s 3ms/step - loss: 3.4738e-04 - accuracy: 1.0000 - val_l
```

```
oss: 0.0498 - val_accuracy: 0.9903
Epoch 13/20
518/518 [==============================] - 1s 3ms/step - loss: 8.5496e-04 - accuracy: 0.9997 - val_l
oss: 0.0428 - val_accuracy: 0.9922
Epoch 14/20
518/518 [==============================] - 1s 3ms/step - loss: 6.4333e-04 - accuracy: 0.9998 - val_l
oss: 0.0432 - val_accuracy: 0.9927
Epoch 15/20
518/518 [==============================] - 1s 3ms/step - loss: 4.7969e-04 - accuracy: 0.9999 - val_l
oss: 0.0444 - val_accuracy: 0.9915
Epoch 16/20
518/518 [==============================] - 1s 3ms/step - loss: 5.3343e-04 - accuracy: 0.9998 - val_l
oss: 0.0494 - val_accuracy: 0.9914
Epoch 17/20
518/518 [==============================] - 1s 3ms/step - loss: 3.4509e-04 - accuracy: 0.9999 - val_l
oss: 0.0491 - val_accuracy: 0.9920
Epoch 18/20
518/518 [==============================] - 1s 3ms/step - loss: 0.0011 - accuracy: 0.9996 - val_loss:
0.0513 - val_accuracy: 0.9904
Epoch 19/20
518/518 [==============================] - 1s 3ms/step - loss: 9.1957e-04 - accuracy: 0.9997 - val_l
oss: 0.0469 - val_accuracy: 0.9912
Epoch 20/20
518/518 [==============================] - 1s 3ms/step - loss: 4.6762e-04 - accuracy: 0.9999 - val_l
oss: 0.0483 - val_accuracy: 0.9915
```

Plot the training loss vs. the epoch for all of the learning rates on one plot. You should see that the lower learning rates are more stable, but converge slower, while with a learning rate that is too high, the gradient descent may fail to move towards weights that decrease the loss function.

In [60]:
```
# TODO 7 one plot showing training loss vs. epoch
# use a different color for each learning rate
plt.figure(figsize=(7,3))

plt.subplot(1,2,1)

nepochs = len(train_loss[0]);
for i in range(len(rates)):
  sns.lineplot(x=np.arange(1,nepochs+1), y=train_loss[i], label=str(rates[i]));

plt.yscale('log')
plt.xlabel('Epochs')
plt.ylabel('Training loss')
```

Out[60]:
```
Text(0, 0.5, 'Training loss')
```