lab_9

November 15, 2023

Shadeeb Hossain

ID : sh7492

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
import requests
url = "http://theremin.music.uiowa.edu/sound files/MIS/Woodwinds/
 ↪sopranosaxophone/001.wav"

response = requests.get(url)
```

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
import librosa

# Specify the filename of the downloaded audio file
fn = "soprano_saxophone.wav"
```

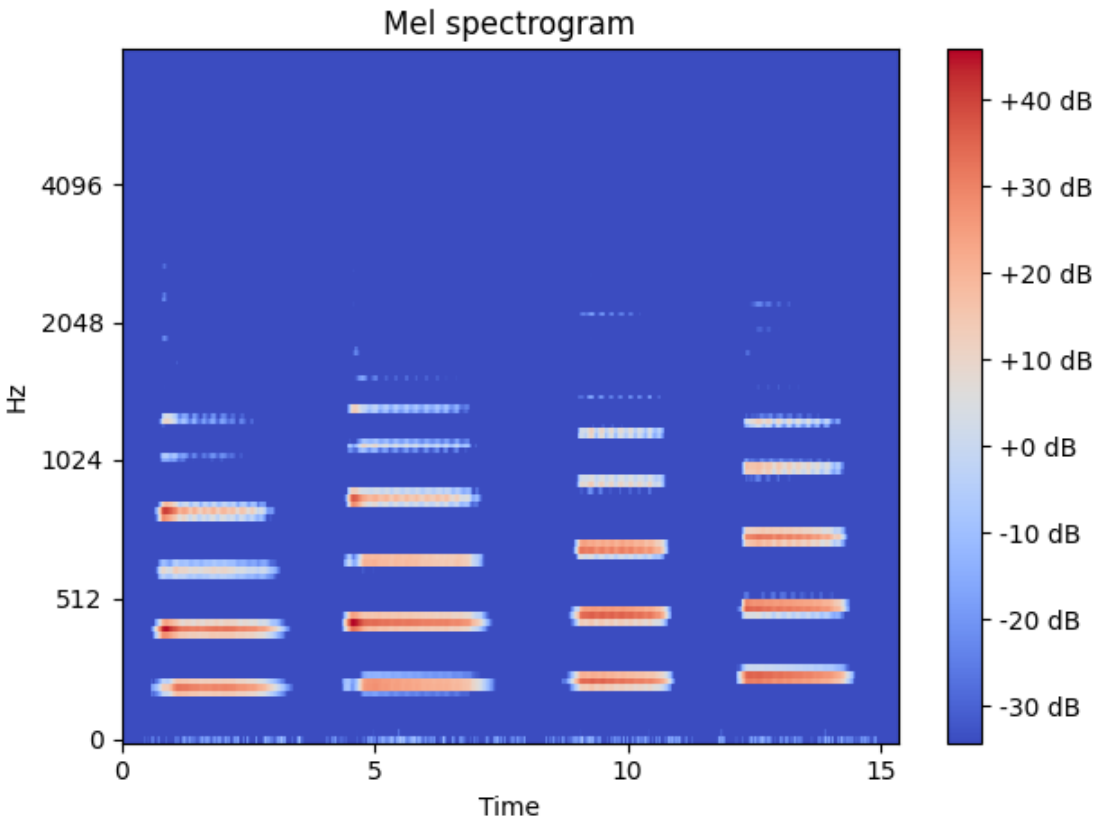
```
# Load the audio file using librosa
y, sr = librosa.load(fn)

# Print the sample rate
print(f"Sample Rate,sr: {sr}")


# Calculate the number of samples
num_samples = len(y)

# Print the result
print(f"Number of Samples,y: {num_samples}")

# y, sr = ...
```

```
Sample Rate,sr: 22050
Number of Samples,y: 338485
```

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.

[ ]:
```
data_dir = 'instrument_dataset/'
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')
```

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?

[ ]:
```python
# TODO 3
import numpy as np

num_training_samples = Xtr.shape[0]
num_test_samples = Xts.shape[0]
num_features = Xtr.shape[1]
num_classes = len(np.unique(ytr))
print(f"Number of Training Samples: {num_training_samples}")
print(f"Number of Test Samples: {num_test_samples}")
print(f"Number of Features for Each Sample: {num_features}")
print(f"Number of Classes (Instruments): {num_classes}")
```

```
Number of Training Samples: 66247
Number of Test Samples: 14904
Number of Features for Each Sample: 120
Number of Classes (Instruments): 10
```

Before continuing, you must scale the training and test data, `Xtr` and `Xts`. Compute the mean and std deviation of each feature in `Xtr` and create a new training data set, `Xtr_scale`, by subtracting the mean and dividing by the std deviation. Also compute a scaled test data set, `Xts_scale` using the mean and std deviation learned from the training data set.

[ ]:
```python
# TODO 4: Scale the training and test matrices
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
Xtr_scale = scaler.fit_transform(Xtr)
Xts_scale = scaler.transform(Xts)
mean_values = scaler.mean_
std_dev_values = scaler.scale_
for feature_idx, (mean, std_dev) in enumerate(zip(mean_values, std_dev_values)):
    print(f"Feature {feature_idx + 1}: Mean = {mean}, Std Dev = {std_dev}")

# Xtr_scale = ...
# Xts_scale = ...
```

```
Feature 1: Mean = 0.000311396769229421, Std Dev = 0.00040639504064489125
Feature 2: Mean = 0.0012518733210245118, Std Dev = 0.0016359706486663481
Feature 3: Mean = 0.005739698137129118, Std Dev = 0.007470869381074944
Feature 4: Mean = 0.003683702080221069, Std Dev = 0.0047717254023178736
Feature 5: Mean = 0.00035321099061597793, Std Dev = 0.000649247639861341
Feature 6: Mean = 0.0008786922331258713, Std Dev = 0.0016046035484330301
Feature 7: Mean = 0.0017007975004097412, Std Dev = 0.0034562010180033
Feature 8: Mean = 0.00182722101337871, Std Dev = 0.003237513938224512
Feature 9: Mean = 0.0015340753467888112, Std Dev = 0.0022034658107452413
Feature 10: Mean = 0.00111329502869382, Std Dev = 0.0014700238157937357
Feature 11: Mean = 0.0002936499993631473, Std Dev = 0.0003513330150665423
Feature 12: Mean = 0.0002758750291924995, Std Dev = 0.0003160420057477934
Feature 13: Mean = 0.0003151550051505951, Std Dev = 0.0004078948575726842
Feature 14: Mean = 0.0012610805276376251, Std Dev = 0.0016403110814976146
```

```
Feature 15: Mean = 0.0057592563440583945, Std Dev = 0.007481516839327951
Feature 16: Mean = 0.003779471094349054, Std Dev = 0.004848610578552358
Feature 17: Mean = 0.001851499820915151, Std Dev = 0.003733581037026553
Feature 18: Mean = 0.002326713224084295, Std Dev = 0.003830305527730013
Feature 19: Mean = 0.0016600519921083645, Std Dev = 0.0021801339004888006
Feature 20: Mean = 0.0010722338301560367, Std Dev = 0.0013330959795030582
Feature 21: Mean = 0.0009381129859177971, Std Dev = 0.0012891168899776657
Feature 22: Mean = 0.0007540062118058538, Std Dev = 0.0010486593984402686
Feature 23: Mean = 0.0002475808081459753, Std Dev = 0.0003033922491731679
Feature 24: Mean = 0.00026747632898197485, Std Dev = 0.0003127869301644979
Feature 25: Mean = 0.00032318303399204854, Std Dev = 0.00041242071153900126
Feature 26: Mean = 0.0012782508780003574, Std Dev = 0.0016503408021406496
Feature 27: Mean = 0.005822657146738516, Std Dev = 0.007510343750979729
Feature 28: Mean = 0.004904532657355201, Std Dev = 0.005937988664408307
Feature 29: Mean = 0.002612424555317668, Std Dev = 0.004110879845944677
Feature 30: Mean = 0.0015485628239274914, Std Dev = 0.001977993186502998
Feature 31: Mean = 0.001062957530846752, Std Dev = 0.001419456870774953
Feature 32: Mean = 0.0007566651452682394, Std Dev = 0.0009765605710845602
Feature 33: Mean = 0.0007405286747822823, Std Dev = 0.003463956063081315
Feature 34: Mean = 0.0006844555898969061, Std Dev = 0.00600561063249584
Feature 35: Mean = 0.0002624929121796849, Std Dev = 0.0020678452122317227
Feature 36: Mean = 0.00027773303999549265, Std Dev = 0.0005237428874371532
Feature 37: Mean = 0.0003732662800138055, Std Dev = 0.0005038739987362332
Feature 38: Mean = 0.0013465690854532876, Std Dev = 0.0017407653442693223
Feature 39: Mean = 0.006265896507429347, Std Dev = 0.007820573458374802
Feature 40: Mean = 0.005564236299702798, Std Dev = 0.00638209984132825
Feature 41: Mean = 0.0019996602176906167, Std Dev = 0.002768667247830578
Feature 42: Mean = 0.0014869223856708441, Std Dev = 0.004382030197972624
Feature 43: Mean = 0.002292761377699467, Std Dev = 0.011587461957492228
Feature 44: Mean = 0.004542458296271169, Std Dev = 0.02302176793270845
Feature 45: Mean = 0.006857455207049204, Std Dev = 0.03841864006147066
Feature 46: Mean = 0.004465203647375712, Std Dev = 0.026479487374836557
Feature 47: Mean = 0.0010931356101381883, Std Dev = 0.006523408312320888
Feature 48: Mean = 0.0005690432202311486, Std Dev = 0.002026578862804948
Feature 49: Mean = 0.0006502461641367714, Std Dev = 0.0012886849488245753
Feature 50: Mean = 0.0017856219262885932, Std Dev = 0.002626621789419288
Feature 51: Mean = 0.007524880644851471, Std Dev = 0.009451404263172486
Feature 52: Mean = 0.006094353097389329, Std Dev = 0.007214769189204942
Feature 53: Mean = 0.004548086026894463, Std Dev = 0.014934801539823883
Feature 54: Mean = 0.01020570938814577, Std Dev = 0.04028376854390529
Feature 55: Mean = 0.019011367720093212, Std Dev = 0.06287884368708248
Feature 56: Mean = 0.03663922936406821, Std Dev = 0.11883798766080533
Feature 57: Mean = 0.049811062133937024, Std Dev = 0.16721848696073452
Feature 58: Mean = 0.0316471126800333, Std Dev = 0.11720811190693925
Feature 59: Mean = 0.007066178639395115, Std Dev = 0.02692912413691749
Feature 60: Mean = 0.003068441624608164, Std Dev = 0.00950201593143194
Feature 61: Mean = 0.0016464531709426513, Std Dev = 0.0028330322370367983
Feature 62: Mean = 0.0034519814328661316, Std Dev = 0.0053345789004732354
```

```
Feature 63: Mean = 0.010592441532703672, Std Dev = 0.014133624787761449
Feature 64: Mean = 0.012760636162184558, Std Dev = 0.027719584538119416
Feature 65: Mean = 0.030823819012017053, Std Dev = 0.08370906416222695
Feature 66: Mean = 0.06792104945677038, Std Dev = 0.18016007289173516
Feature 67: Mean = 0.08668014235498148, Std Dev = 0.18753625560723985
Feature 68: Mean = 0.10457362285247745, Std Dev = 0.2035580964018604
Feature 69: Mean = 0.13837530262467346, Std Dev = 0.2934528407242002
Feature 70: Mean = 0.10160402592986294, Std Dev = 0.2555266542924642
Feature 71: Mean = 0.022841940872902703, Std Dev = 0.06486945962720905
Feature 72: Mean = 0.009643340763889304, Std Dev = 0.020873328231636578
Feature 73: Mean = 0.003952720762866292, Std Dev = 0.0053218946438314246
Feature 74: Mean = 0.007716699931419308, Std Dev = 0.010514128255061284
Feature 75: Mean = 0.020872456904079694, Std Dev = 0.029385483771856943
Feature 76: Mean = 0.05634364518095409, Std Dev = 0.12801483616407935
Feature 77: Mean = 0.12297187877184687, Std Dev = 0.2395486485067916
Feature 78: Mean = 0.19423590631617776, Std Dev = 0.32097600665261333
Feature 79: Mean = 0.2389646133997495, Std Dev = 0.3805925492166691
Feature 80: Mean = 0.21918523807885518, Std Dev = 0.2952241566598412
Feature 81: Mean = 0.22635999128363002, Std Dev = 0.3326480660198443
Feature 82: Mean = 0.15007437522353892, Std Dev = 0.2701685423155584
Feature 83: Mean = 0.034802059858287215, Std Dev = 0.06607241891303556
Feature 84: Mean = 0.01660917977725553, Std Dev = 0.023793583821865943
Feature 85: Mean = 0.008266726823506751, Std Dev = 0.010174682724942574
Feature 86: Mean = 0.0181379252983375, Std Dev = 0.024995564195450674
Feature 87: Mean = 0.05731863597057501, Std Dev = 0.08743870963376203
Feature 88: Mean = 0.17658229791812846, Std Dev = 0.2900223146703629
Feature 89: Mean = 0.31971895913020815, Std Dev = 0.44451410335645836
Feature 90: Mean = 0.3609601370306811, Std Dev = 0.4018780014712252
Feature 91: Mean = 0.37756816457567965, Std Dev = 0.5267775288620964
Feature 92: Mean = 0.3626789976088831, Std Dev = 0.5676862627798128
Feature 93: Mean = 0.2920707490142066, Std Dev = 0.4324942654291931
Feature 94: Mean = 0.17774842156191115, Std Dev = 0.3028861563525236
Feature 95: Mean = 0.044826708708531074, Std Dev = 0.07688163272543634
Feature 96: Mean = 0.024093617642807645, Std Dev = 0.029670862828872907
Feature 97: Mean = 0.013773887406600023, Std Dev = 0.015443944793854905
Feature 98: Mean = 0.03459150581230184, Std Dev = 0.042537075886046334
Feature 99: Mean = 0.14843414160501167, Std Dev = 0.21472479635808076
Feature 100: Mean = 0.40841336040418913, Std Dev = 0.49538452070068584
Feature 101: Mean = 0.5475656355842535, Std Dev = 0.6257584950100844
Feature 102: Mean = 0.47479012234450846, Std Dev = 0.6667480650420692
Feature 103: Mean = 0.32970266036149426, Std Dev = 0.510159530598439
Feature 104: Mean = 0.21392692454020235, Std Dev = 0.3500075901616581
Feature 105: Mean = 0.1226250861433303, Std Dev = 0.19932285263342478
Feature 106: Mean = 0.06949770875196622, Std Dev = 0.11775136029067026
Feature 107: Mean = 0.026151252752579852, Std Dev = 0.03494978050142828
Feature 108: Mean = 0.0189834190067075, Std Dev = 0.022207713948765858
Feature 109: Mean = 0.02016771748913776, Std Dev = 0.02153661488894984
Feature 110: Mean = 0.059734250213526435, Std Dev = 0.06908714930681291
```

```
Feature 111: Mean = 0.33450028243484436, Std Dev = 0.4173303038634999
Feature 112: Mean = 0.6978061891490636, Std Dev = 0.7629307220935556
Feature 113: Mean = 0.552986077520673, Std Dev = 0.7694415325410582
Feature 114: Mean = 0.2678544245418904, Std Dev = 0.4379444615989003
Feature 115: Mean = 0.10430745438525815, Std Dev = 0.17527920747487555
Feature 116: Mean = 0.049508112901865686, Std Dev = 0.08227447508081245
Feature 117: Mean = 0.03161837672212778, Std Dev = 0.04692959253690212
Feature 118: Mean = 0.025430424198947834, Std Dev = 0.03281257117423299
Feature 119: Mean = 0.019596308733505013, Std Dev = 0.023538764418767455
Feature 120: Mean = 0.020463797057093357, Std Dev = 0.022607686990297263
```

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

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

[ ]:
```
# TODO 5 clear session
import tensorflow as tf
from tensorflow.keras import layers, models
tf.keras.backend.clear_session()
```

[ ]:
```
# TODO 6: construct the model
nh = 256
model = models.Sequential()
model.add(layers.Dense(nh, activation='sigmoid', input_shape=(num_features,)))
model.add(layers.Dense(num_classes, activation='softmax'))
```

[ ]:
```
# TODO 7:  Print the model summary
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               30976

 dense_1 (Dense)             (None, 10)                2570

=================================================================
Total params: 33546 (131.04 KB)
Trainable params: 33546 (131.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

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

[ ]:
```
# TODO 8
from tensorflow.keras import optimizers
optimizer = optimizers.Adam(learning_rate=0.001)
model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',  # Assuming a␣
 ↪classification problem
              metrics=['accuracy'])
model.summary()

# opt = ...
# model.compile(...)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 256)               30976

 dense_1 (Dense)             (None, 10)                2570

=================================================================
Total params: 33546 (131.04 KB)
Trainable params: 33546 (131.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

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%.

[ ]:
```
# TODO 9
from tensorflow.keras.callbacks import Callback

tf.keras.backend.clear_session()
nh = 256
model = models.Sequential()
model.add(layers.Dense(nh, activation='sigmoid', input_shape=(num_features,)))
model.add(layers.Dense(num_classes, activation='softmax'))

# Create an instance of the Adam optimizer with a learning rate of 0.001
optimizer = optimizers.Adam(learning_rate=0.001)

# Compile the model
model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
```

```
              metrics=['accuracy'])

# Create an instance of your custom callback class
my_callback_instance = MyCustomCallback(target_accuracy=0.99)

# Fit the model
history = model.fit(
    Xtr_scale, ytr,
    epochs=10,
    batch_size=100,
    validation_data=(Xts_scale, yts),
    callbacks=[my_callback_instance]
)

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(Xts_scale, yts)

# Print the final accuracy
print(f"Final Test Accuracy: {test_accuracy * 100:.2f}%")
```

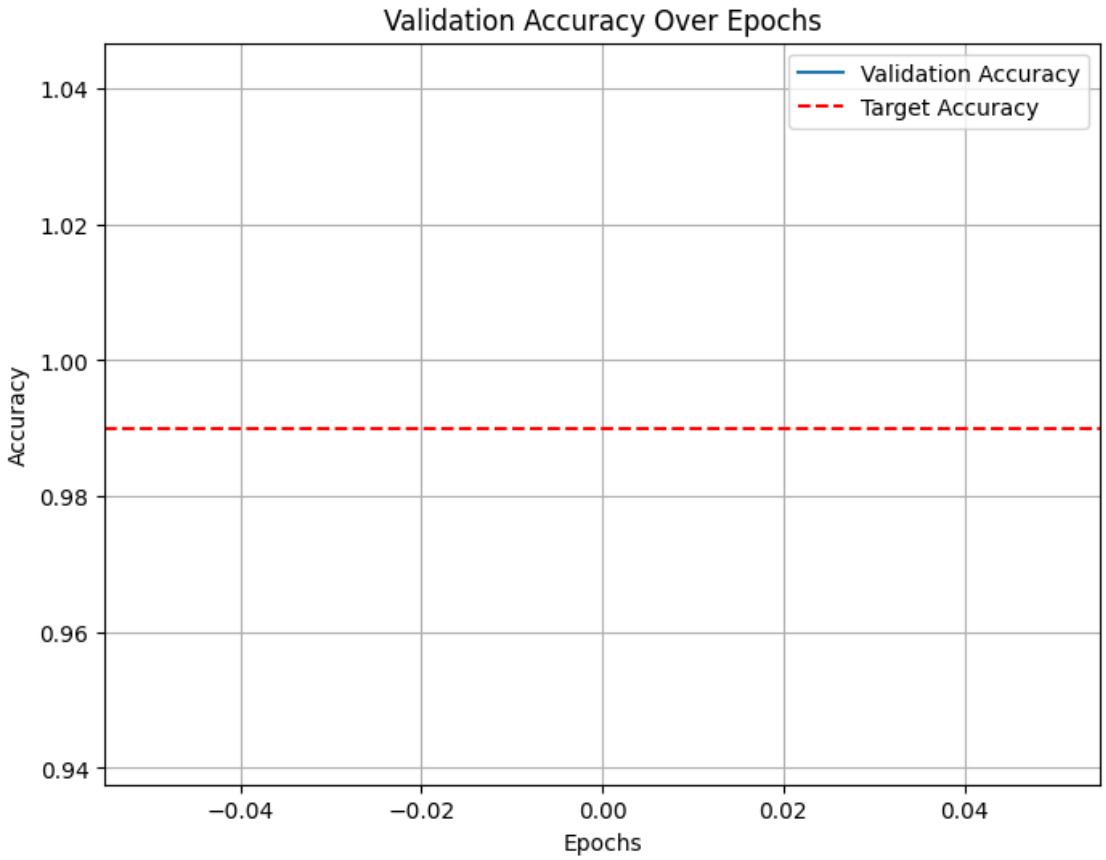
```
Epoch 1/10
663/663 [==============================] - 4s 4ms/step - loss: 0.3612 -
accuracy: 0.9020 - val_loss: 0.1729 - val_accuracy: 0.9621
Epoch 2/10
663/663 [==============================] - 2s 4ms/step - loss: 0.1019 -
accuracy: 0.9752 - val_loss: 0.0950 - val_accuracy: 0.9781
Epoch 3/10
663/663 [==============================] - 3s 4ms/step - loss: 0.0597 -
accuracy: 0.9858 - val_loss: 0.0637 - val_accuracy: 0.9847
Epoch 4/10
663/663 [==============================] - 3s 5ms/step - loss: 0.0422 -
accuracy: 0.9897 - val_loss: 0.0492 - val_accuracy: 0.9865
Epoch 5/10
663/663 [==============================] - 2s 4ms/step - loss: 0.0319 -
accuracy: 0.9919 - val_loss: 0.0404 - val_accuracy: 0.9899
Epoch 6/10
663/663 [==============================] - 2s 4ms/step - loss: 0.0250 -
accuracy: 0.9936 - val_loss: 0.0482 - val_accuracy: 0.9845
Epoch 7/10
663/663 [==============================] - 2s 3ms/step - loss: 0.0208 -
accuracy: 0.9945 - val_loss: 0.0363 - val_accuracy: 0.9890
Epoch 8/10
654/663 [============================>.] - ETA: 0s - loss: 0.0171 - accuracy:
0.9957
Reached target accuracy (99.00%), stopping training!
663/663 [==============================] - 2s 3ms/step - loss: 0.0171 -
accuracy: 0.9957 - val_loss: 0.0280 - val_accuracy: 0.9918
466/466 [==============================] - 1s 2ms/step - loss: 0.0280 -
```

```
accuracy: 0.9918
Final Test Accuracy: 99.18%
```

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.

[ ]:
```python
# TODO 10
import matplotlib.pyplot as plt
history = model.fit(
    Xtr_scale, ytr,
    epochs=10,
    batch_size=100,
    validation_data=(Xts_scale, yts),
    callbacks=[my_callback_instance]
)
plt.figure(figsize=(8, 6))
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.axhline(y=0.99, color='r', linestyle='--', label='Target Accuracy')
plt.title('Validation Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```

```
Epoch 1/10
652/663 [============================>.] - ETA: 0s - loss: 0.0144 - accuracy:
0.9964
Reached target accuracy (99.00%), stopping training!
663/663 [==============================] - 3s 5ms/step - loss: 0.0145 -
accuracy: 0.9963 - val_loss: 0.0233 - val_accuracy: 0.9921
```

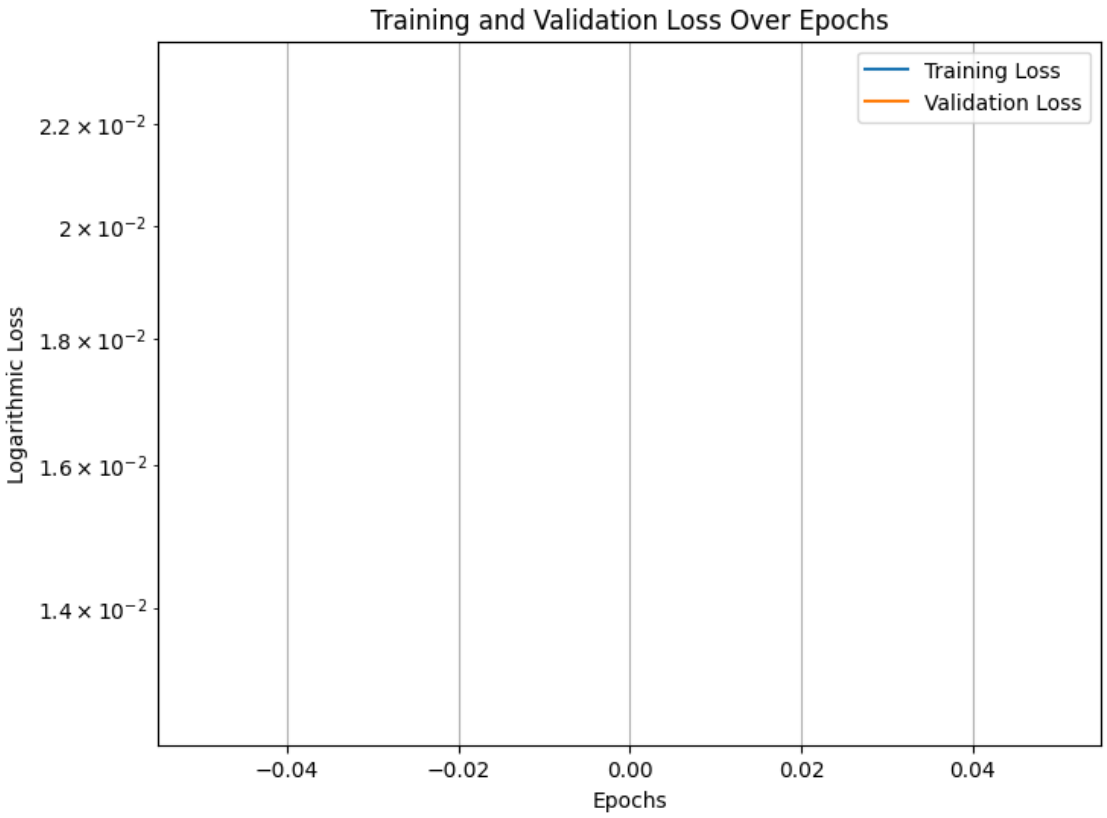


Plot the loss values saved in the `hist.history` dictionary. You should see that the loss is steadily decreasing. Use the `semilogy` plot.

[ ]:
```
# TODO 11
import matplotlib.pyplot as plt
history = model.fit(
    Xtr_scale, ytr,
    epochs=10,
    batch_size=100,
    validation_data=(Xts_scale, yts),
    callbacks=[my_callback_instance]
)
plt.figure(figsize=(8, 6))
plt.semilogy(history.history['loss'], label='Training Loss')
plt.semilogy(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Logarithmic Loss')
plt.legend()
plt.grid(True)
```

```
plt.show()
```

```
Epoch 1/10
661/663 [============================>.] - ETA: 0s - loss: 0.0127 - accuracy:
0.9968
Reached target accuracy (99.00%), stopping training!
663/663 [==============================] - 4s 6ms/step - loss: 0.0127 -
accuracy: 0.9968 - val_loss: 0.0231 - val_accuracy: 0.9921
```



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

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

# TODO 12
```

```
rates = [0.01, 0.001, 0.0001]
batch_size = 100
loss_hist = []
for lr in rates:
    print(f"\nTraining model with learning rate: {lr}")
    tf.keras.backend.clear_session()
    model = models.Sequential()
    model.add(layers.Dense(nh, activation='sigmoid',␣
 ↪input_shape=(num_features,)))
    model.add(layers.Dense(num_classes, activation='softmax'))
    optimizer = optimizers.Adam(learning_rate=lr)
    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    history = model.fit(
        Xtr_scale, ytr,
        epochs=20,
        batch_size=batch_size,
        validation_data=(Xts_scale, yts)
    )
    loss_hist.append(history.history['val_loss'])
plt.figure(figsize=(8, 6))
for i, lr in enumerate(rates):
    plt.plot(loss_hist[i], label=f'LR={lr}')

plt.title('Validation Loss Over Epochs for Different Learning Rates')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

```
Training model with learning rate: 0.01
Epoch 1/20
663/663 [==============================] - 4s 6ms/step - loss: 0.1054 -
accuracy: 0.9681 - val_loss: 0.0544 - val_accuracy: 0.9811
Epoch 2/20
663/663 [==============================] - 3s 4ms/step - loss: 0.0294 -
accuracy: 0.9902 - val_loss: 0.0483 - val_accuracy: 0.9840
Epoch 3/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0202 -
accuracy: 0.9933 - val_loss: 0.0279 - val_accuracy: 0.9899
Epoch 4/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0171 -
accuracy: 0.9943 - val_loss: 0.0515 - val_accuracy: 0.9811
Epoch 5/20
```

```
663/663 [==============================] - 2s 3ms/step - loss: 0.0162 -
accuracy: 0.9948 - val_loss: 0.0284 - val_accuracy: 0.9899
Epoch 6/20
663/663 [==============================] - 3s 4ms/step - loss: 0.0139 -
accuracy: 0.9953 - val_loss: 0.0417 - val_accuracy: 0.9874
Epoch 7/20
663/663 [==============================] - 4s 5ms/step - loss: 0.0132 -
accuracy: 0.9957 - val_loss: 0.0643 - val_accuracy: 0.9822
Epoch 8/20
663/663 [==============================] - 2s 4ms/step - loss: 0.0132 -
accuracy: 0.9956 - val_loss: 0.0404 - val_accuracy: 0.9875
Epoch 9/20
663/663 [==============================] - 4s 6ms/step - loss: 0.0143 -
accuracy: 0.9957 - val_loss: 0.0456 - val_accuracy: 0.9873
Epoch 10/20
663/663 [==============================] - 7s 10ms/step - loss: 0.0095 -
accuracy: 0.9970 - val_loss: 0.0684 - val_accuracy: 0.9830
Epoch 11/20
663/663 [==============================] - 6s 9ms/step - loss: 0.0106 -
accuracy: 0.9966 - val_loss: 0.0363 - val_accuracy: 0.9891
Epoch 12/20
663/663 [==============================] - 3s 4ms/step - loss: 0.0107 -
accuracy: 0.9967 - val_loss: 0.0694 - val_accuracy: 0.9826
Epoch 13/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0101 -
accuracy: 0.9970 - val_loss: 0.0309 - val_accuracy: 0.9906
Epoch 14/20
663/663 [==============================] - 3s 5ms/step - loss: 0.0101 -
accuracy: 0.9968 - val_loss: 0.0389 - val_accuracy: 0.9900
Epoch 15/20
663/663 [==============================] - 3s 5ms/step - loss: 0.0076 -
accuracy: 0.9975 - val_loss: 0.0359 - val_accuracy: 0.9917
Epoch 16/20
663/663 [==============================] - 2s 4ms/step - loss: 0.0100 -
accuracy: 0.9971 - val_loss: 0.1172 - val_accuracy: 0.9773
Epoch 17/20
663/663 [==============================] - 3s 4ms/step - loss: 0.0100 -
accuracy: 0.9971 - val_loss: 0.0378 - val_accuracy: 0.9905
Epoch 18/20
663/663 [==============================] - 4s 6ms/step - loss: 0.0098 -
accuracy: 0.9971 - val_loss: 0.1310 - val_accuracy: 0.9724
Epoch 19/20
663/663 [==============================] - 7s 10ms/step - loss: 0.0096 -
accuracy: 0.9972 - val_loss: 0.0497 - val_accuracy: 0.9885
Epoch 20/20
663/663 [==============================] - 3s 5ms/step - loss: 0.0086 -
accuracy: 0.9977 - val_loss: 0.0492 - val_accuracy: 0.9891
```

```
Training model with learning rate: 0.001
Epoch 1/20
663/663 [==============================] - 3s 4ms/step - loss: 0.3625 -
accuracy: 0.9007 - val_loss: 0.2276 - val_accuracy: 0.9231
Epoch 2/20
663/663 [==============================] - 2s 3ms/step - loss: 0.1020 -
accuracy: 0.9752 - val_loss: 0.1165 - val_accuracy: 0.9600
Epoch 3/20
663/663 [==============================] - 4s 5ms/step - loss: 0.0596 -
accuracy: 0.9861 - val_loss: 0.0712 - val_accuracy: 0.9800
Epoch 4/20
663/663 [==============================] - 3s 4ms/step - loss: 0.0422 -
accuracy: 0.9894 - val_loss: 0.0479 - val_accuracy: 0.9891
Epoch 5/20
663/663 [==============================] - 2s 4ms/step - loss: 0.0319 -
accuracy: 0.9917 - val_loss: 0.0418 - val_accuracy: 0.9894
Epoch 6/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0252 -
accuracy: 0.9936 - val_loss: 0.0410 - val_accuracy: 0.9877
Epoch 7/20
663/663 [==============================] - 4s 5ms/step - loss: 0.0213 -
accuracy: 0.9941 - val_loss: 0.0339 - val_accuracy: 0.9904
Epoch 8/20
663/663 [==============================] - 7s 10ms/step - loss: 0.0174 -
accuracy: 0.9955 - val_loss: 0.0262 - val_accuracy: 0.9915
Epoch 9/20
663/663 [==============================] - 4s 6ms/step - loss: 0.0150 -
accuracy: 0.9960 - val_loss: 0.0258 - val_accuracy: 0.9918
Epoch 10/20
663/663 [==============================] - 3s 5ms/step - loss: 0.0131 -
accuracy: 0.9964 - val_loss: 0.0224 - val_accuracy: 0.9928
Epoch 11/20
663/663 [==============================] - 3s 4ms/step - loss: 0.0115 -
accuracy: 0.9970 - val_loss: 0.0335 - val_accuracy: 0.9877
Epoch 12/20
663/663 [==============================] - 3s 5ms/step - loss: 0.0102 -
accuracy: 0.9971 - val_loss: 0.0298 - val_accuracy: 0.9896
Epoch 13/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0092 -
accuracy: 0.9977 - val_loss: 0.0244 - val_accuracy: 0.9910
Epoch 14/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0085 -
accuracy: 0.9978 - val_loss: 0.0231 - val_accuracy: 0.9916
Epoch 15/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0076 -
accuracy: 0.9982 - val_loss: 0.0238 - val_accuracy: 0.9919
Epoch 16/20
663/663 [==============================] - 3s 4ms/step - loss: 0.0070 -
```

```
accuracy: 0.9981 - val_loss: 0.0237 - val_accuracy: 0.9911
Epoch 17/20
661/663 [============================>.] - ETA: 0s - loss: 0.0064 - accuracy:
0.9983
```

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.

[ ]:
```python
# TODO 13
import matplotlib.pyplot as plt


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

for lr in rates:
    print(f"\nTraining model with learning rate: {lr}")
    tf.keras.backend.clear_session()
    model = models.Sequential()
    model.add(layers.Dense(nh, activation='sigmoid',␣
 ↪input_shape=(num_features,)))
    model.add(layers.Dense(num_classes, activation='softmax'))
    optimizer = optimizers.Adam(learning_rate=lr)
    model.compile(optimizer=optimizer,
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    history = model.fit(
        Xtr_scale, ytr,
        epochs=20,
        batch_size=batch_size,
        validation_data=(Xts_scale, yts)
    )
    loss_hist.append(history.history['val_loss'])
plt.figure(figsize=(8, 6))
for i, lr in enumerate(rates):
    plt.plot(loss_hist[i], label=f'LR={lr}')

plt.title('Validation Loss Over Epochs for Different Learning Rates')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

```
Training model with learning rate: 0.01
```

```
Epoch 1/20
663/663 [==============================] - 3s 4ms/step - loss: 0.1068 -
accuracy: 0.9671 - val_loss: 0.2831 - val_accuracy: 0.9181
Epoch 2/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0276 -
accuracy: 0.9909 - val_loss: 0.0308 - val_accuracy: 0.9892
Epoch 3/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0207 -
accuracy: 0.9931 - val_loss: 0.0475 - val_accuracy: 0.9839
Epoch 4/20
663/663 [==============================] - 3s 5ms/step - loss: 0.0170 -
accuracy: 0.9945 - val_loss: 0.0513 - val_accuracy: 0.9810
Epoch 5/20
663/663 [==============================] - 3s 4ms/step - loss: 0.0175 -
accuracy: 0.9945 - val_loss: 0.0239 - val_accuracy: 0.9923
Epoch 6/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0127 -
accuracy: 0.9960 - val_loss: 0.0446 - val_accuracy: 0.9847
Epoch 7/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0158 -
accuracy: 0.9953 - val_loss: 0.0480 - val_accuracy: 0.9857
Epoch 8/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0132 -
accuracy: 0.9961 - val_loss: 0.0541 - val_accuracy: 0.9819
Epoch 9/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0118 -
accuracy: 0.9963 - val_loss: 0.0416 - val_accuracy: 0.9879
Epoch 10/20
663/663 [==============================] - 3s 5ms/step - loss: 0.0133 -
accuracy: 0.9958 - val_loss: 0.0208 - val_accuracy: 0.9934
Epoch 11/20
663/663 [==============================] - 3s 4ms/step - loss: 0.0098 -
accuracy: 0.9970 - val_loss: 0.0436 - val_accuracy: 0.9860
Epoch 12/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0148 -
accuracy: 0.9956 - val_loss: 0.0293 - val_accuracy: 0.9925
Epoch 13/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0076 -
accuracy: 0.9976 - val_loss: 0.0566 - val_accuracy: 0.9873
Epoch 14/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0095 -
accuracy: 0.9969 - val_loss: 0.0301 - val_accuracy: 0.9920
Epoch 15/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0068 -
accuracy: 0.9979 - val_loss: 0.0536 - val_accuracy: 0.9890
Epoch 16/20
663/663 [==============================] - 3s 5ms/step - loss: 0.0095 -
accuracy: 0.9970 - val_loss: 0.0451 - val_accuracy: 0.9893
```

```
Epoch 17/20
663/663 [==============================] - 3s 4ms/step - loss: 0.0085 -
accuracy: 0.9977 - val_loss: 0.0555 - val_accuracy: 0.9862
Epoch 18/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0122 -
accuracy: 0.9965 - val_loss: 0.0493 - val_accuracy: 0.9887
Epoch 19/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0075 -
accuracy: 0.9978 - val_loss: 0.0745 - val_accuracy: 0.9847
Epoch 20/20
663/663 [==============================] - 2s 3ms/step - loss: 0.0052 -
accuracy: 0.9985 - val_loss: 0.0494 - val_accuracy: 0.9889
```

[ ]:

[ ]: