TECHIN 513 – Lab 6
Deep Neural Networks

Lab Instructions

1. Please approach the course instructor or grader for assistance if you need help.
2. We highly recommend working in groups of **2 members**. When collaborating with students in other groups, please do not share code but only discuss the relevant concepts and implementation methods.
3. Please document your code well by **using appropriate comments, variable names, spacing, indentation, docstrings**, etc. Please refer to TECHIN 509 for more information.
4. The starter code is not binding on you. Feel free to modify it as you wish. Everything is fine so long as you are getting the correct results.
5. Please upload the .ipynb file to canvas. Use Markdown cells appropriately to answer the questions. Each team only needs to submit one report.
6. Please enter the names of all the team members in your Jupyter notebook so that you do not lose credit for your work.

Lab Submission Requirements

In your notebook, briefly describe your results and discuss the problems you encountered, the solutions that you came up with. Also, answer the questions asked in each section. Make sure to include the code of all tasks in code cells.

Preparation

You will need a few packages such as TensorFlow for tasks in this lab.

If you would like to perform this lab locally, setting up a virtual environment for this lab would reduce the chance of creating conflicts among dependencies. Please refer to TECHIN 509, or https://docs.python.org/3/library/venv.html, or https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html on how to setup virtual environment.

Please follow the steps (https://www.tensorflow.org/install) to install TensorFlow locally.

Alternatively, Google Colab has the dependencies well-configured, and should be ready for your use.

Lab Objectives

This lab is meant to give you an introduction to working with feature engineering and deep neural network using Python.

This lab could also include machine learning tasks for you to get familiar with some machine learning tools used for classifying audio data, and to highlight the importance of signal processing for such tasks. Specifically, you can design a classifier that identifies birds from the Pacific Northwest region. For this you will be using short bird call/chatter sounds uploaded by the users

of https://xeno-canto.org/ compiled by the Cornell Lab of Ornithology (https://birdnet.cornell.edu/). The dataset provided consists recordings of over 240 types of birds but for the purpose of this lab we will stick to 22 common birds seen in the Pacific Northwest region: American Robin, Crow, Song Sparrow, Dark-eyed Junco, Black-capped Chickadee, Mallard, Northern Flicker, Spotted Towhee, House Finch, Anna's Hummingbird, Goose, Red-winged Blackbird, Great Blue Heron, Steller's Jay, American Goldfinch, White-crowned Sparrow, Red-breasted Nuthatch, Red-tailed Hawk, Bewick'sWren, Common Raven, Double-crested Cormorant and House Sparrow.

The starter code for this lab contains code for loading the dataset and demonstrate how neural networks are implemented in Python using Tensorflow. Feel free to deviate from the starter code (e.g., Pytorch and Keras) and implement different neural network architectures or different signal preprocessing pipelines to improve your classification accuracy as much as possible. However, you should not overfit your model or improve your model performance using the testing dataset.

Dataset Description

For this lab, we will use the Cornell Birdcall Identification dataset[1] that contains sounds from over 240 types of birds. A preprocessed version of this dataset with approximately 2000 bird sounds from 22 different birds is provided to you. You can download the dataset using this link [2] (note that you should use your UW email to download the dataset). Unzip the downloaded file and carefully configure the path to the data directory in your code. The unzipped file size of the dataset is approximately 5 GB. The dataset has already been split into a training and a testing set, which you can find in the train and test folders, respectively.

You will train your classifier on the training set and evaluate its performance on the testing set. The train set contains 80% of all bird sounds. The test set contains the remaining 20%. Each bird sound is 30s long, sampled with a sampling frequency of 44100 Hz. Besides the audio recordings in the train and test folder, tables with meta data are provided for the training and testing set (train.csv and test.csv, respectively). These meta data tables can be loaded in Python as pandas.DataFrames.

Task 1: Load Dataset and Listen to Bird Sound (6 points)

1. Use the function load_dataset() (with parameters transform = None) to load training and testing datasets. We use signal.resample_poly() to downsample our data from 44100Hz to 22050Hz due to the RAM constraints on a typical laptop. The code for loading the training dataset is already provided in the starter code. You need to write the code for loading the testing dataset. load_dataset() returns a tuple (data, labels), where data contains the audio time series organized in a 2D numpy array and labels contains the labels (One-hot encoder of bird_code) for each audio sample. Each label consists of an array of size 22 with 21 zeros and 1 one at the position corresponding to the bird type. Each row of data is a single bird recording and the corresponding row in labels is the corresponding bird. That is, the audio time series and label of the i-th recording can be accessed by audio = data[i] and label = labels[i].
2. Plot the time series of a single recording using plt.plot() and play the bird sound using the

[1] https://www.kaggle.com/compe22ons/birdsong-recogni2on/data
[2]

IPython.display.Audio() function from Lab 1 or other valid options. You can display recording at i = 0, 150, 250, 350. The x-axis of your plot should be in seconds. Make sure to label both x- and y-axis properly. Print the label of the corresponding recording. Note that the sampling frequency is 22050 Hz, because load dataset() downsampled the recording.

Task 2: A Naïve Deep Learning Implementation (4 points)

Our first approach for building a classifier is to disregard any signal processing and let deep learning do all the work. We will later see that this does not perform well and add a signal processing stage to our pipeline that will significantly improve the performance. For now, our classification pipeline looks as follows.

The core of this classifier is a fully connected Multilayer Perceptron (MLP). This neural network architecture is composed of a collection of neurons that are organized in layers, whereby the neurons of one layer are connected to all neurons of the next layer. An example of such a neural network is shown in Slide 8 of lecture.

Introduction to Tensorflow and Simple Neural Network Implementation

At a first glance, implementing a neural network architecture like the one we discussed in class seems very daunting. Luckily, Python provides many libraries that make implementing neural networks easy. One such library is tensorflow, which has been developed by Google Brain and was initially released in 2015. Specifically, we will use the tensorflow.keras.models module that can be used to build and customize even complex neural network architectures with just a few lines of code. The input to our neural network is the 20s long bird sound time series sampled with a frequency of 22050 Hz. As the input layer has one unit for each input feature, the input layer of our architecture consists of 20x22050 = 441000 units. To design a neural network, we first initialize the model by calling

model = model.Sequential().

Now we can add fully connected layers to the model by calling

model.add(layers.Dense(100, activation='sigmoid')).

The first parameter in Dense specifies the number of neurons for this layer (i.e., 100 in this case). The second parameter specifies the non-linearity used by each neuron. A common non-linearity is the sigmoid activation function as discussed in class.

Other common non-linearities are the rectified linear unit (ReLU), or hyperbolic tangent (tanh) function. We will use ReLU for our layers except for the output layer. Because our classification problem has 22 possibilities, our output layer should have 22 neurons that return values between 0 and 1, depending on how close the sound is to the bird type. We can implement this layer analogously to the previous layer:

model.add(layers.Dense(22, activation='sigmoid')).

To build the model we use

model.build(data_train[0].shape),

where data_train is our training set. This will also tell our model the number of units of our input layer. Finally, to plot a summary of the model, we can call model.summary().

Task 2a (2 points)

Add at least two additional Dense layers to the network architecture in the starter code. You are free to choose the number of neurons, and activation function in each layer.

To train the model, we need two more lines of code. First, we need to compile the model by calling

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']).

You do not need to worry about the optimizer and loss parameter at this point. The parameter metric=['accuracy'] indicates that we want to achieve a classification accuracy, i.e., the percentage of correctly classified samples, as high as possible. Finally, the actual training is performed by calling

history = model.fit(data_train, labels_train, epochs=10, validation_data=(data_test, labels_test), shuffle=True),

where data_train and labels_train are training dataset and labels, and data_test and labels_test are testing dataset and labels, respectively. The epoch parameter indicates how many times the entire training dataset is used for training the model. After each epoch, we compute the accuracy on the testing dataset. When you run those two lines of code, you will see a printout that contains, among other things, the classification accuracy on the train and test set after each epoch.

Task 2b (2 points)

Run the model fitting and describe (in a separate markdown cell), how the classification accuracy on the training and testing set changes throughout the model training. Plot training and testing set accuracy as a function of epoch number and report the final accuracy on the testing set. (The code for that is already provided and all you need is to execute it.) Assume, you have a new bird sound sample that the model has not seen before. How likely do you think it is that the model identifies the right bird? Comment on this in a markdown cell.

Task 3: Changing the Signal Representation

The spectrogram is one form of time-frequency representation and shows how the spectrum of an audio signal changes over time. It is computed via the Short-Time Fourier Transform (STFT) given by

$$X[m,k] = \sum_{n=0}^{N-1} x[n]w[n-mT]e^{-j\frac{2\pi}{N}nk} \quad (1)$$

where $w[n]$ is a window of length N and T is the window hop size. The indices m and k in Eq. (1) represent time and frequency, respectively. Intuitively, Eq. (1) states that we break our discrete time signal $x[n]$ into a series of segments, multiply each segment by a window function $w[n]$, and finally compute the DFT of each windowed segment. This process is also illustrated in Fig. 3. To finally obtain the spectrogram, we take the magnitude square of the STFT:

$$S[m,k] = |X[m,k]|^2 \quad (2)$$

Since the spectrogram is a 2-dimensional function of time and frequency, we can visualize it like an image, as shown in Fig. 1, where the horizontal axis indicates time and the vertical axis frequency[3].

[3] The common spectrogram representation includes some additional normalization constants, which we have ignored here for simplicity. Furthermore, following a common practice in machine learning, we will normalize

Task 3a: Spectrogram of Bird Sounds

1. Read the function spec() that computes the spectrogram of a single audio signal. The function takes as inputs the audio signal x and the window length of the STFT N. Assume a window hop size of $T = N$. That is, adjacent segments do not overlap. For the window function we use a simple rectangular window. The implementation includes the following steps:
    a. split the signal x into K segments of length N.
    b. compute the FFT of each segment and only store the frequency points from $k = 0$ to $N/2$ (corresponding to the DTFT frequency from 0 to π.).
    c. Compute the magnitude square according to Eq. (2).
    d. Compute the normalized spectrogram

$$\tilde{S}[m,k] = \frac{S[m,k]}{\max\{S[m,k]\}} \tag{3}$$

2. Run the code cell that computes the spectrogram of a single bird sound and compare it with numpy's spectrogram function used in spec2(). The two spectrograms you get should look identical. (Hint: You can use the function np.fft.fft() and might find the function np.split() helpful. With those functions it is possible to write an implementation without any for loops)

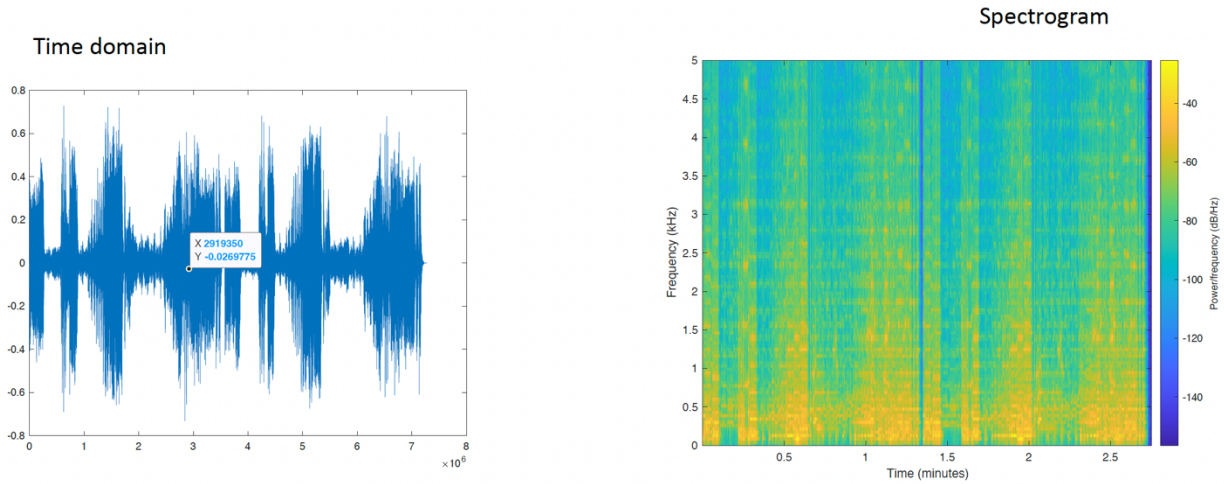
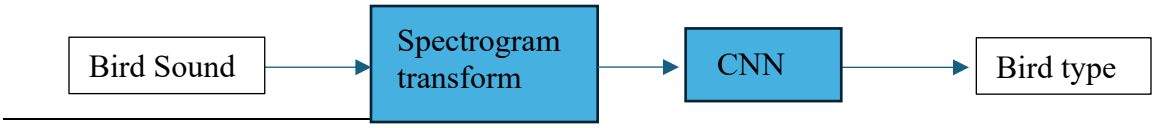


Figure 1: Spectrogram demonstrating how frequency (right) varies over time for audio (Beethoven Symphony No. 5) in time domain (left).

Task 4: Using Spectrogram and CNN to Identify Bird Sounds

In this final section, we use the spectrogram representation to design a classifier to identify bird sounds. Since the spectrogram is a 2-dimensional array (like an image) we replace the architecture in Task 2 by a convolutional neural network (CNN). Please refer to the lecture slides for the basic architecture of CNNs. Now, our classification pipeline will look as follows:



S[m,k] to have a maximum magnitude of 1 rendering the need for normalization constants unnecessary.

Task 4a: CNN Implementation (10 points)

1. Load the train and test set using the load dataset() function with the parameter transform='spec' (you only need to run the corresponding code cell in the starter code).
2. Build the CNN architecture. An initial architecture with one convolutional layer has already been provided in the starter code. Add at least one more convolutional layer followed by a MaxPooling layer to the architecture. The important input parameters of the Conv2D layer are
    a. filters: number of convolutional kernels
    b. kernel size: size of each kernel
    c. activation: The activation function that is used for the output of the convolutional layer. You should use the 'ReLU' function.

    The MaxPooling layer only takes a pool size that is equivalent to the kernel size of the convolutional layer.
3. Run the next code cell of the starter notebook to train your architecture. As in Task 2, the printout will tell you the accuracy of the classifier on the training and testing set after each epoch.
4. As in Task 2, plot the accuracy on the train and test set as a function of epoch.
5. Change the parameters of your CNN architecture (number of layers, kernel size, etc.) to get as high of an accuracy on the testing set as possible. Describe the observations you make (What changes seem to improve/decrease the test set accuracy? How does the accuracy on train and test set change throughout the training process (i.e., with increasing number of epochs)?). You should be able to get an accuracy of at least 30% to 32% on the testing set.