1 Brief Introduction

Can a 2-hidden-layer MLP do a good job classifying musical instrument sounds? Let’s find out!

2 Introduction

For beginner guitar players, it’s sometimes difficult to tell apart the sound of steel strings vs nylon strings on the guitar. In this article, I’ll walk you through some easy steps to build a Machine Learning model to classify the two aforemtioned types of sound.

You can try a live DEMO via:

https://nqhoang2077-steel-nylon-streamlitapp-streamlit-app-8sd4cc.streamlit.app/

3 Import libraries & Setup constants

The first step is to install all required libraries. Even though torchaudio could handle audio, they lack support for some media formats. That’s why we need two additional sound libraries, namely libosa and soundfile. Our main data source is YouTube, and pytube allows easy and fast audio extraction from the social media platform.

!pip install torchaudio librosa soundfile pytube torchsummary matplotlib pandas

|████████████████████████████████| 56 kB 3.6 MB/s eta 0:00:01

!conda install libsoundfile

**import** torch

**import** torchaudio

**import** librosa

**import** pandas **as** pd

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**import** soundfile **as** sf

**import** threading

**from** glob **import** glob

**import** os

Use Google Drive if you need to store your data there.

*# from google.colab import drive*

*# drive.mount('/content/drive')*

Mounted at /content/drive

For simplicity, I will use 8000 Hz as the default sampling rate. This helps training faster on modest hardware. Also, I’d like to segment each YouTube audio clip into chunks of 5-second clip for training. This help us enrich our dataset and simplify our network architecture.

TARGET\_SR = 8\_000

CLASSES = ["nylon", "steel"]

SEGMENT\_DURATION = 5 *# seconds*

RANDOM\_SEED = 42

TRAIN\_SIZE = .95

torch.random.manual\_seed(RANDOM\_SEED)

<torch.\_C.Generator at 0x7faa2c165c90>

Then, we will create some folders to store our sound files.

ROOT\_DIR = "./"

DATA\_DIR = f"{ROOT\_DIR}data/" *# Google Drive & Colab*

**for** subfolder **in** ["raw", "segments"]:

**for** cls **in** CLASSES:

new\_dir = f"{DATA\_DIR}{subfolder}/{cls}"

**if** **not** os.path.exists(new\_dir):

os.makedirs(new\_dir)

SEGMENT\_DIR = f"{DATA\_DIR}segments/"

RAW\_CLIP\_PATH = f"{DATA\_DIR}raw/"

This function helps get the best device possible for training.

**def** get\_best\_torch\_device():

**if** torch.cuda.is\_available():

device = "cuda"

**else**:

device = "cpu"

print(f"Using device {device}")

**return** device

device = get\_best\_torch\_device()

Using device cuda

4 Collect Data

4.1 Download audio from YouTube

These are the clips that I handpicked from YouTube. They are solo guitar recordings and were recorded in a professional studio. To watch any of them, just add the youtube url prefix. For example: **“foIPN-T7RGo” ➡️ “youtube.com/watch?v=foIPN-T7RGo”**

steel\_clips = ["foIPN-T7RGo","10ATKnZLg9c","IP8vBL5Q8Ac"]

nylon\_clips = ["qgb-bdEEI-M","qXwvz-nTiog","6jQ34uTmA9s"]

Next, we define our function to download and extract audio from a YouTube url:

**from** pytube **import** YouTube

**def** download\_youtube\_mp3(link, output\_dir):

*"""*

*Download and extract audio from a clip from youtube*

*"""*

yt=YouTube(f"youtube.com/watch?v={link}")

t=yt.streams.filter(only\_audio=True).first().download(output\_dir, link + ".mp3")

print(f"Downloaded YouTube Audio from: {link}")

Each clip is over 60 minutes long, which could take a long time to download. To accelerate, we will create a downloading thread for each clip and download all clips simultaneously.

download\_thread\_list = []

**for** link **in** steel\_clips:

new\_thread = threading.Thread(target=download\_youtube\_mp3, args=(link, RAW\_CLIP\_PATH + "steel"))

download\_thread\_list.append(new\_thread)

**for** link **in** nylon\_clips:

new\_thread = threading.Thread(target=download\_youtube\_mp3, args=(link, RAW\_CLIP\_PATH + "nylon"))

download\_thread\_list.append(new\_thread)

print("Download Raw Clips starting...")

*# start each thread*

**for** thread **in** download\_thread\_list:

thread.start()

*# wait for all to finish*

**for** thread **in** download\_thread\_list:

thread.join()

*# successfully excecuted*

print("Download Raw Clips finished!")

Download Raw Clips starting...

Downloaded YouTube Audio from: foIPN-T7RGo

Downloaded YouTube Audio from: 6jQ34uTmA9s

Downloaded YouTube Audio from: qgb-bdEEI-M

Downloaded YouTube Audio from: qXwvz-nTiog

Downloaded YouTube Audio from: 10ATKnZLg9c

Downloaded YouTube Audio from: IP8vBL5Q8Ac

Download Raw Clips finished!

4.2 Segmentize into 5-second clips

Now, let’s create some function to segment each audio clip into segments of 5 second long.

**def** segmentize\_signal(signal, sr, dur):

*"""*

*Segmentize the 1-d signal (mono) to a list of clips with custom duration (dur).*

*"""*

seg\_len = dur \* sr

*# calculate number of segments*

no\_segs = len(signal) // seg\_len

*# truncate input signal to have length divisiable by seg\_len*

trunc\_len = int(no\_segs \* seg\_len)

*# split equally*

**return** np.split(signal[:trunc\_len], no\_segs)

**def** save\_audio(signal, sr, output\_dir, filename):

output\_path = os.path.join(output\_dir, filename)

*# torchaudio.save(output\_path, signal, sr)*

*# print(output\_path, sr)*

sf.write(output\_path, signal, sr)

**def** segment\_audio\_file(audio\_path, output\_dir, target\_sr=TARGET\_SR, segment\_duration=SEGMENT\_DURATION):

print(f"Processing raw clip: {audio\_path}")

signal, \_ = librosa.load(audio\_path, sr=target\_sr, mono=True)

*# signal, target\_sr = librosa.load(audio\_path,sr=None, mono=True)*

print(f"\tLoaded clip from disk")

segments\_list = segmentize\_signal(signal, target\_sr, segment\_duration)

print(f"\tSegmented clip into {len(segments\_list)} segments")

**for** seg\_idx, seg **in** enumerate(segments\_list):

seg\_name = f"{audio\_path.split('/')[-1][:-4]}\_{seg\_idx}.wav"

save\_audio(seg, target\_sr, output\_dir, seg\_name)

print(f"\tSegments are saved completely")

Next, we use threading to segmentize all clips at the same time. Beware that if your system has less than 32GB of RAM, this could cause the system to freeze and run out of memory. In such case, please modify the code before do it sequentially (i.e. without threading)

thread\_list = []

**for** cls **in** CLASSES:

*# get all raw files from subfolders*

raw\_audio\_paths = glob(f"{RAW\_CLIP\_PATH}{cls}/\*mp3")

**for** audio\_path **in** raw\_audio\_paths:

output\_dir = f"{SEGMENT\_DIR}{cls}"

new\_thread = threading.Thread(target=segment\_audio\_file, args=(audio\_path, output\_dir))

thread\_list.append(new\_thread)

print("Segmentation starting...")

*# start each thread*

**for** thread **in** thread\_list:

thread.start()

*# wait for all to finish*

**for** thread **in** thread\_list:

thread.join()

*# successfully excecuted*

print("Segmentation finished!")

Segmentation starting...

Processing raw clip: /workspace/data/raw/nylon/qXwvz-nTiog.mp3

Processing raw clip: /workspace/data/raw/nylon/6jQ34uTmA9s.mp3

Processing raw clip: /workspace/data/raw/nylon/qgb-bdEEI-M.mp3

Processing raw clip: /workspace/data/raw/steel/IP8vBL5Q8Ac.mp3

Processing raw clip: /workspace/data/raw/steel/foIPN-T7RGo.mp3

/opt/conda/lib/python3.7/site-packages/librosa/util/decorators.py:88: UserWarning: PySoundFile failed. Trying audioread instead.

return f(\*args, \*\*kwargs)

/opt/conda/lib/python3.7/site-packages/librosa/util/decorators.py:88: UserWarning: PySoundFile failed. Trying audioread instead.

return f(\*args, \*\*kwargs)

Processing raw clip: /workspace/data/raw/steel/10ATKnZLg9c.mp3

/opt/conda/lib/python3.7/site-packages/librosa/util/decorators.py:88: UserWarning: PySoundFile failed. Trying audioread instead.

return f(\*args, \*\*kwargs)

Loaded clip from disk

Segmented clip into 648 segments

Segments are saved completely

Loaded clip from disk

Segmented clip into 742 segments

Segments are saved completely

Loaded clip from disk

Segmented clip into 1230 segments

Segments are saved completely

Loaded clip from disk

Segmented clip into 1251 segments

Segments are saved completely

Loaded clip from disk

Segmented clip into 1427 segments

Segments are saved completely

Loaded clip from disk

Segmented clip into 2647 segments

Segments are saved completely

Segmentation finished!

5 Dataset & Dataloader

PyTorch manages data through two types of classes: Dataset and Data;oader. Dataset could be thought of as an iterator that allows us to access each individual data point. And, Dataloader is a way to efficiently load data in batch, which is useful for mini-batch training. For more detailed description, read here: https://pytorch.org/tutorials/beginner/basics/data\_tutorial.html

5.1 Create annotations

Before creating our own dataset class, we need to have a csv file to describe our training / val / test sets.

This annotation dataframe stores the paths to each audio sample and its label:

annotation\_dict = {"audio\_path": [], "label": []}

**for** label, cls **in** enumerate(CLASSES):

wav\_dirs = f"{SEGMENT\_DIR}{cls}/\*wav"

audio\_path\_list = glob(wav\_dirs)

count\_audio\_files = len(audio\_path\_list)

label\_list = [label] \* count\_audio\_files

annotation\_dict["audio\_path"] += audio\_path\_list

annotation\_dict["label"] += label\_list

annotation\_df = pd.DataFrame.from\_dict(annotation\_dict)

annotation\_df.tail()

|  | **audio\_path** | **label** |
| --- | --- | --- |
| **7940** | ./data/segments/steel/foIPN-T7RGo\_575.wav | 1 |
| **7941** | ./data/segments/steel/10ATKnZLg9c\_545.wav | 1 |
| **7942** | ./data/segments/steel/10ATKnZLg9c\_1035.wav | 1 |
| **7943** | ./data/segments/steel/10ATKnZLg9c\_602.wav | 1 |
| **7944** | ./data/segments/steel/IP8vBL5Q8Ac\_1146.wav | 1 |

The data is quite enormously for an average system. That’s why I seperated the training data set to full, half, quarter, and one eighth. This allows me to build and test model fast (by using a smaller training dataset). When I find something that works well, I can then use a larger training dataset to improve the training.

train\_df\_full = annotation\_df.sample(frac=TRAIN\_SIZE, random\_state=RANDOM\_SEED)

val\_df = annotation\_df.drop(train\_df\_full.index, axis=0)

*# make smaller train datasets for quick experimentations*

train\_df\_half = train\_df\_full.sample(frac=1/2, random\_state=RANDOM\_SEED)

train\_df\_quarter = train\_df\_full.sample(frac=1/4, random\_state=RANDOM\_SEED)

train\_df\_1eight = train\_df\_full.sample(frac=1/8, random\_state=RANDOM\_SEED)

We have 4816 samples of NYLON, and 3129 of STEEL

annotation\_df["label"].value\_counts()

0 4816

1 3129

Name: label, dtype: int64

Finally, let’s write them to CSV files for later use.

df\_list = [train\_df\_full, train\_df\_half, train\_df\_quarter, train\_df\_1eight, val\_df]

df\_names = ["train\_df\_full", "train\_df\_half", "train\_df\_quarter", "train\_df\_1eight", "val\_df"]

**for** df\_name, df\_content **in** zip(df\_names, df\_list):

df\_content.to\_csv(f"{DATA\_DIR}{df\_name}.csv", index=False)

5.2 Dataset class

We create GuitarSoundDataset which inherets Dataset from PyTorch. This class holds the annotation that we created earlier and helps us access and preprocess each individual input and label.

To create this class, I took inspiration from this awesome Deep Learning for Audio channel: https://www.youtube.com/watch?v=iCwMQJnKk2c&t=1s&ab\_channel=ValerioVelardo-TheSoundofAI

**from** torch.utils.data **import** Dataset

**class** GuitarSoundDataset(Dataset):

**def** \_\_init\_\_(self,

annotations\_file,

transformation,

target\_sample\_rate,

num\_samples,

device,

audio\_col="audio\_path",

label\_col="label"):

self.annotations = pd.read\_csv(annotations\_file)

self.device = device

**if** transformation:

self.transformation = transformation.to(self.device)

**else**:

self.transformation = None

self.target\_sample\_rate = target\_sample\_rate

self.num\_samples = num\_samples

self.audio\_col = audio\_col

self.label\_col = label\_col

**def** \_\_len\_\_(self):

**return** len(self.annotations)

**def** \_\_getitem\_\_(self, index):

audio\_sample\_path = self.\_\_get\_audio\_sample\_path(index)

label = self.\_\_get\_audio\_sample\_label(index)

signal, sr = torchaudio.load(audio\_sample\_path)

**if** signal.dim() < 2:

signal = signal[None, :]

signal = signal.to(self.device)

signal, sr = self.preprocess\_signal(signal, sr)

**if** self.transformation:

signal = self.transformation(signal)

**return** signal, label

**def** preprocess\_signal(self, signal, sr):

signal = self.\_\_resample\_if\_necessary(signal, sr)

signal = self.\_\_mix\_down\_if\_necessary(signal)

signal = self.\_\_cut\_if\_necessary(signal)

signal = self.\_\_right\_pad\_if\_necessary(signal)

**return** signal, sr

**def** \_\_cut\_if\_necessary(self, signal):

**if** signal.shape[1] > self.num\_samples:

signal = signal[:, :self.num\_samples]

**return** signal

**def** \_\_right\_pad\_if\_necessary(self, signal):

length\_signal = signal.shape[1]

**if** length\_signal < self.num\_samples:

num\_missing\_samples = self.num\_samples - length\_signal

last\_dim\_padding = (0, num\_missing\_samples)

signal = torch.nn.functional.pad(signal, last\_dim\_padding)

**return** signal

**def** \_\_resample\_if\_necessary(self, signal, sr):

**if** sr != self.target\_sample\_rate:

resampler = torchaudio.transforms.Resample(sr, self.target\_sample\_rate).to(self.device)

signal = resampler(signal)

**return** signal

**def** \_\_mix\_down\_if\_necessary(self, signal):

**if** signal.shape[0] > 1:

signal = torch.mean(signal, dim=0, keepdim=True)

**return** signal

**def** \_\_get\_audio\_sample\_path(self, index):

path = self.annotations.iloc[index, :][self.audio\_col]

**return** path

**def** \_\_get\_audio\_sample\_label(self, index):

label = self.annotations.iloc[index, :][self.label\_col]

**return** torch.tensor(label, dtype=torch.float)

5.3 DataLoader

**from** torch.utils.data **import** DataLoader

**def** create\_data\_loader(dataset, batch\_size):

dataset\_loader = DataLoader(dataset, batch\_size=batch\_size)

**return** dataset\_loader

Mel Spectrogram transforms our signal from time-domain into frequency-domain, which helps not only human but also computers to understand the characteristic of sound input better. Thus, we need to transform each audio input into mel spec before feeding it into the neural network.

mel\_spectrogram = torchaudio.transforms.MelSpectrogram(

sample\_rate=TARGET\_SR,

n\_fft=1024,

hop\_length=512,

n\_mels=64

)

train\_dataset = GuitarSoundDataset(

annotations\_file =f"{DATA\_DIR}train\_df\_half.csv",

transformation = mel\_spectrogram,

target\_sample\_rate = TARGET\_SR,

num\_samples = TARGET\_SR \* SEGMENT\_DURATION,

device = device)

print(f"There are {len(train\_dataset)} samples in the TRAIN dataset.")

val\_dataset = GuitarSoundDataset(f"{DATA\_DIR}val\_df.csv",

transformation = mel\_spectrogram,

target\_sample\_rate = TARGET\_SR,

num\_samples = TARGET\_SR \* SEGMENT\_DURATION,

device = device)

print(f"There are {len(val\_dataset)} samples in the VAL dataset.")

There are 3774 samples in the TRAIN dataset.

There are 397 samples in the VAL dataset.

We will take one sample out to find out the exact input shape for our neural network

signal\_sample, \_ = val\_dataset[0]

signal\_sample.shape

torch.Size([1, 64, 79])

6 Build Model

6.1 Training Loop

Because the training and validating loops are pretty basic, I don’t delve into these code too much. The official tutorial is where I took inspiration from: https://pytorch.org/tutorials/beginner/blitz/cifar10\_tutorial.html

**def** compute\_accuracy(preds, target):

\_preds = preds.detach().cpu().numpy()

\_target = target.detach().cpu().numpy()

**return** np.mean(\_preds.squeeze().round() == \_target.squeeze())

**def** train\_single\_epoch(model, data\_loader, loss\_fn, optimiser, device):

size = len(data\_loader.dataset)

train\_losses = []

train\_accs = []

model.train(True)

**for** batch, (input, target) **in** enumerate(data\_loader):

input, target = input.to(device), target.to(device)

*# calculate loss*

preds = model(input)

loss = loss\_fn(preds.squeeze(), target.squeeze())

train\_losses.append(loss.item())

*# backpropagate error and update weights*

optimiser.zero\_grad()

loss.backward()

optimiser.step()

*# calculate accuracy*

acc = compute\_accuracy(preds, target)

train\_accs.append(acc)

**return** np.mean(train\_losses), np.mean(train\_accs)

**def** validate(model, data\_loader, loss\_fn, device):

*# model.train(False)*

val\_losses = []

val\_accs = []

**with** torch.inference\_mode():

**for** input, target **in** data\_loader:

input, target = input.to(device), target.to(device)

*# calculate loss*

preds = model(input)

loss = loss\_fn(preds.squeeze(), target.squeeze())

val\_losses.append(loss.item())

*# calculate acc*

acc = compute\_accuracy(preds, target)

val\_accs.append(acc)

**return** np.mean(val\_losses), np.mean(val\_accs)

**def** save\_model(model, model\_dir):

torch.save(model.state\_dict(), model\_dir)

**def** train(model, train\_dataloader, test\_dataloader, loss\_fn, optimiser, device, epochs, save\_best=True, model\_dir="bestmodel.pth"):

train\_losses = []

train\_accs = []

val\_losses = []

val\_accs = []

**for** i **in** range(epochs):

*# training*

train\_loss, train\_acc = train\_single\_epoch(model, train\_dataloader, loss\_fn, optimiser, device)

*# val*

val\_loss, val\_acc = validate(model, test\_dataloader, loss\_fn, device)

print(f"Epoch {i+1} | train loss: {train\_loss:.5f}, train acc: {train\_acc:.3%} | val loss: {val\_loss:.5f}, val acc: {val\_acc:.3%}")

*# save best val acc*

**if** save\_best **and** len(val\_losses) > 0 **and** val\_acc > np.max(val\_accs):

*# save model*

print("-> Best Model found! Saving to disk...")

save\_model(model, model\_dir)

*# update losses*

train\_losses.append(train\_loss)

val\_losses.append(val\_loss)

train\_accs.append(train\_acc)

val\_accs.append(val\_acc)

print("Finished training")

**return** train\_losses, train\_accs, val\_losses, val\_accs

**def** plot\_model(model\_history):

train\_losses, train\_accs, val\_losses, val\_accs = model\_history

*# Plot Loss*

plt.plot(range(len(train\_losses)), train\_losses, label='Training Loss')

plt.plot(range(len(train\_losses)), val\_losses, label='Validation Loss')

*# Add in a title and axes labels*

plt.title('Training and Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend(loc="upper left")

plt.show()

*# Plot Acc*

plt.plot(range(len(train\_accs)), train\_accs, label='Training Acc')

plt.plot(range(len(train\_accs)), val\_accs, label='Validation Acc')

*# Add in a title and axes labels*

plt.title('Training and Validation Acc')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend(loc="upper left")

plt.show()

**def** describe\_model\_stats(model\_history):

train\_losses, train\_accs, val\_losses, val\_accs = model\_history

history = {"train\_losses": train\_losses, "train\_accs": train\_accs, "val\_losses": val\_losses, "val\_accs": val\_accs}

print(pd.DataFrame.from\_dict(history).describe())

6.2 MLP Model Building: 2 hidden layers with ReLu Activation

I define a simple MLP with 2 hidden fully connected layers with relu activation. The final output is then taken by sigmoid to produce probabily prediction.

**from** torch **import** nn

**from** torchsummary **import** summary

**class** MLPNetwork(nn.Module):

**def** \_\_init\_\_(self):

super().\_\_init\_\_()

self.flatten = nn.Flatten()

self.linear = nn.Sequential(

nn.Linear(1 \* 64 \* 79, 256), *# I got the number (1 \* 64 \* 79) as input size from the code above*

nn.ReLU(),

nn.Linear(256, 128),

nn.ReLU(),

nn.Linear(128, 1),

)

**def** forward(self, input\_data):

x = self.flatten(input\_data)

logits = self.linear(x)

predictions = torch.sigmoid(logits)

**return** predictions

*# return x*

**if** \_\_name\_\_ == "\_\_main\_\_":

model2 = MLPNetwork()

summary(model2.to(device), (1, 64, 79))

----------------------------------------------------------------

Layer (type) Output Shape Param #

================================================================

Flatten-1 [-1, 5056] 0

Linear-2 [-1, 256] 1,294,592

ReLU-3 [-1, 256] 0

Linear-4 [-1, 128] 32,896

ReLU-5 [-1, 128] 0

Linear-6 [-1, 1] 129

================================================================

Total params: 1,327,617

Trainable params: 1,327,617

Non-trainable params: 0

----------------------------------------------------------------

Input size (MB): 0.02

Forward/backward pass size (MB): 0.04

Params size (MB): 5.06

Estimated Total Size (MB): 5.13

----------------------------------------------------------------

Audio input is complex, with an audio sample of 5-second long at 8000 Hz sampling rate, we have an input of 5056 already.

And, this simple MLP model already has 1.3+ millions params.

Now, let’s create a folder to store our trained params.

MODEL\_DIR = f"{ROOT\_DIR}weights/"

**if** **not** os.path.exists(MODEL\_DIR):

os.makedirs(MODEL\_DIR)

Then, define some hyper params for training and create dataloader for each training and validation dataset

BATCH\_SIZE = 128

EPOCHS = 15

LEARNING\_RATE = 0.001

train\_dataloader = create\_data\_loader(train\_dataset, BATCH\_SIZE)

val\_dataloader = create\_data\_loader(val\_dataset, BATCH\_SIZE)

Now, let’s train our model!

MODEL\_SAVE\_PATH = f"{MODEL\_DIR}model\_mlp1.pth"

print(f"Best models will saved to: {MODEL\_DIR} (based on val acc)")

model1 = MLPNetwork()

**if** os.path.exists(MODEL\_SAVE\_PATH):

model1.load\_state\_dict(torch.load(MODEL\_SAVE\_PATH, map\_location=torch.device(device)))

model1 = model1.to(device)

*# initialise loss funtion + optimiser*

loss\_fn = nn.BCELoss()

optimiser = torch.optim.Adam(model1.parameters(),

lr=LEARNING\_RATE)

*# train model*

history\_model1 = train(model1, train\_dataloader, val\_dataloader, loss\_fn, optimiser, device, EPOCHS, save\_best=True, model\_dir=MODEL\_SAVE\_PATH)

Best models will saved to: ./weights/ (based on val acc)

Epoch 1 | train loss: 18.60054, train acc: 74.353% | val loss: 18.41207, val acc: 79.943%

Epoch 2 | train loss: 16.17105, train acc: 80.607% | val loss: 14.15756, val acc: 82.287%

-> Best Model found! Saving to disk...

Epoch 3 | train loss: 16.04773, train acc: 79.716% | val loss: 22.44626, val acc: 72.251%

Epoch 4 | train loss: 15.91658, train acc: 80.841% | val loss: 24.57104, val acc: 72.446%

Epoch 5 | train loss: 15.68584, train acc: 81.720% | val loss: 26.59615, val acc: 71.274%

Epoch 6 | train loss: 17.09794, train acc: 80.188% | val loss: 15.84533, val acc: 81.671%

Epoch 7 | train loss: 15.53885, train acc: 82.014% | val loss: 15.38519, val acc: 80.364%

Epoch 8 | train loss: 12.86597, train acc: 84.409% | val loss: 19.71215, val acc: 75.931%

Epoch 9 | train loss: 12.15642, train acc: 85.247% | val loss: 13.57315, val acc: 81.926%

Epoch 10 | train loss: 11.81812, train acc: 84.438% | val loss: 12.96833, val acc: 81.145%

Epoch 11 | train loss: 10.44457, train acc: 86.577% | val loss: 10.41160, val acc: 82.677%

-> Best Model found! Saving to disk...

Epoch 12 | train loss: 8.54932, train acc: 88.063% | val loss: 7.47481, val acc: 84.826%

-> Best Model found! Saving to disk...

Epoch 13 | train loss: 7.18438, train acc: 88.478% | val loss: 6.82821, val acc: 83.263%

Epoch 14 | train loss: 5.54964, train acc: 89.264% | val loss: 4.69765, val acc: 84.405%

Epoch 15 | train loss: 2.19403, train acc: 89.026% | val loss: 1.49113, val acc: 84.075%

Finished training

describe\_model\_stats(history\_model1)

plot\_model(history\_model1)

train\_losses train\_accs val\_losses val\_accs

count 15.000000 15.000000 15.000000 15.000000

mean 12.388064 0.836627 14.304709 0.798988

std 4.770562 0.042507 7.314085 0.046287

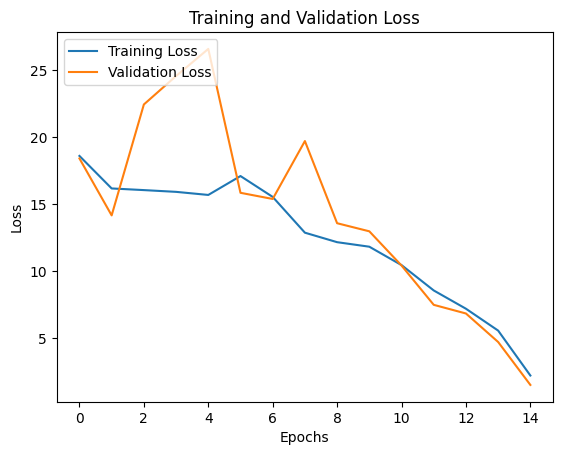
min 2.194033 0.743532 1.491129 0.712740

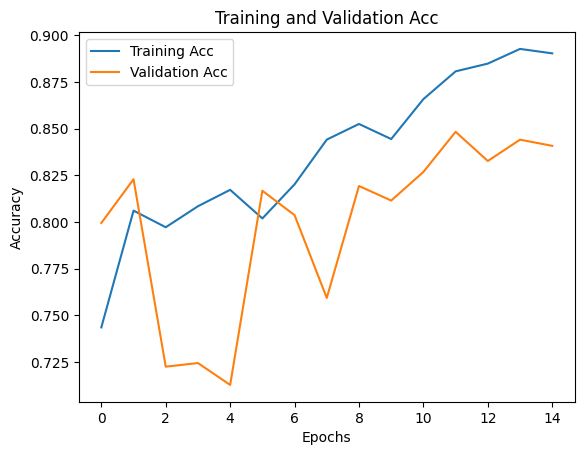
25% 9.496942 0.807237 8.943201 0.779372

50% 12.865967 0.844086 14.157557 0.816707

75% 15.982155 0.873198 19.062112 0.829703

max 18.600537 0.892641 26.596150 0.848257





7 Conclusion

With this simple architecture, we already achieve an acceptable accuracy of round 81%. Not bad for our first try.

In the live demo, I actually used a more complicated CNN model which achieves over 90% validation accuracy. The training was made possibly by running on a GPU cloud with RTX3090, AMD EPYC Cpu and 83GB of RAM. You can [try them out here - RunPod](https://runpod.io/?ref=c7xd5dcy) (my affiliate link)

8 Acknowledgment

I would like to thank [AI VIETNAM](https://www.facebook.com/aivietnam.edu.vn) for providing the basic knowledge about mathematics and machine learning model, and [Valerio Velardo](https://www.youtube.com/@ValerioVelardoTheSoundofAI) for his tutorials about working with sound data. Last but not least, I give my thanks to all the artists whose clips I used in this model which is for educational purpose.