Healthy Listening

Building a Wearable Edge Microphone

DataSci251 - Spring 2023

James Meyer, He Shi, Charlee Stefanski

Abstract

Visits to the doctor's office play an important role in the healthcare process, but the unfortunate reality is that not all medical information that may be necessary to properly diagnose a patient can be observed or quantified in a short in-person visit. Given these limitations, wearable edge devices have the potential to add significant value to the medical field, allowing for longer-term data collection on patients that can assist medical professionals in making more accurate diagnoses. One example of this would be the symptom of coughing. Coughs come in many different varieties, and both the tone and frequency of coughs may vary depending on the particular ailment, however it can be difficult to precisely measure these within the confines of an office visit, with medical professionals often forced to rely on their patient's memory or attempts to recreate their specific symptoms. In this project, our goal will be to develop an "edge microphone", capable of continuously recording audio from a patient, successfully detecting coughs, and transmitting the recordings back to a cloud server to be used by medical professionals during their diagnostic efforts.

Process Overview

The first step will be to identify a high-quality dataset that can be used to train and validate a high-accuracy audio classification model. Various models will then be trained on the dataset in the cloud to take advantage of GPU computing resources. Once an optimal model is identified it will then be optimized for the edge before exporting the weights and parameters for use in inference on the edge device. Due to hardware resource constraints, for the purpose of this project a local virtual machine will be used to emulate the computing limitations of a physical edge device. The final step will be to set up the programs and network necessary to record audio on the edge device, run inference on it, and transmit it back to a long-term cloud storage device for future analysis

The Data

Dataset & Data Creation

Due to the challenges associated with collecting and labeling the thousands of individual audio files that would be required to accurately train an audio classification model, it was determined that the best course of action would be to find a publicly available dataset that could serve as the basis for our training. After a period of research we discovered the ESC-50: Dataset for Environmental Sound Classification[1]. The ESC-50 is a dataset containing 2,000 five-second WAV audio files, each of which fall into one of 50 different labels spread over 5 categories including Animals, Natural Sounds, Human Sounds, Domestic Sounds, and Urban Noises. The audio files are distributed evenly across all 50 labels, with each label containing 40 unique examples. All the data files in the ESC-50 dataset were manually collected and sorted from the larger Freesound.org database which includes a vast array of Creative Commons Licensed (CCL) audio files. We were fortunate in that the dataset already included examples of our target label, namely “coughing”, meaning we were able to avoid any arduous data collection during the training and validation process.

For testing our model’s ability to conduct inference on the edge, we wanted to better replicate the exact types of audio that would be generated by the edge device. As a result, we decided to collect and label our own audio samples using the same USB webcam hardware that would be part of the final design. In total we collected a test dataset of 120 five-second audio files, totaling 10 minutes of audio. Given that our ultimate interest lies in whether a particular audio file is identified as a cough or not, rather than labeling the test data according to the 50 labels available in the original dataset we opted to simply separate the data into either “cough”, which would be assigned a value of 1, or “not a cough”, which would receive a value of 0. Of the 120 files in the test dataset, 20 were labeled as cough. This simple breakdown allowed us to get a sense of the final accuracy and precision of our models for this application.

[1] K. J. Piczak. ESC: Dataset for Environmental Sound Classification. *Proceedings of the 23rd Annual ACM Conference on Multimedia*, Brisbane, Australia, 2015.

Understanding Audio Data

Unlike image data, where information is divided into individual pixels which each have a number or set of numbers which capture the color for that pixel, audio signals are broken down in samples which represent the value of the audio signal at a given moment in time, allowing a continuous signal to be turned into a discrete set of samples. A common sampling frequency is 44.1 kHz, which means that the audio signal is sampled 44,100 times per second. In the WAV audio format, each sample is stored in 16 bits, totaling 441 kB of data for a five-second audio file.

In order to train a model based on the audio dataset we had available, we have to convert this stream of 16-bit samples into a format that can be fed into the neural network of the models we are training. The most common solution to this problem is to create what is known as a spectrogram. A spectrogram is a visual representation of the strength and frequency of an audio signal across time. The Y-axis tracks the frequency of the signal, measure in hertz (Hz) or cycles per second, while the color at a given frequency displays the decibels (dB) or strength of the signal with warmer colors representing higher decibels and lowder sounds, while cooler colors represent lower decibels and softer sounds. In the figures below, examples of spectrograms have been provided and it becomes immediately clear how models trained on such images of audio data would be able to recognize key features in both the frequency and strength that would allow for accurate audio classification.

Fig. 1 - Coughing

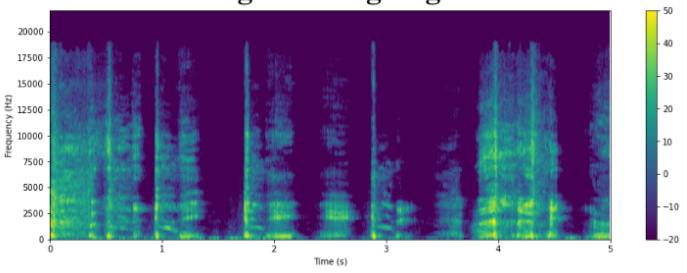
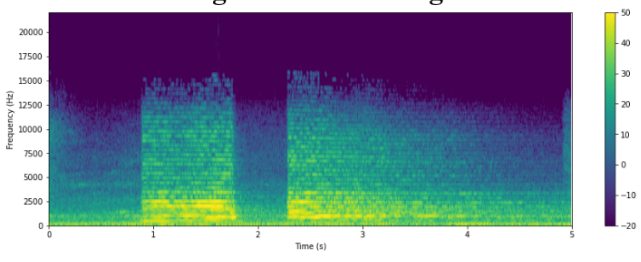


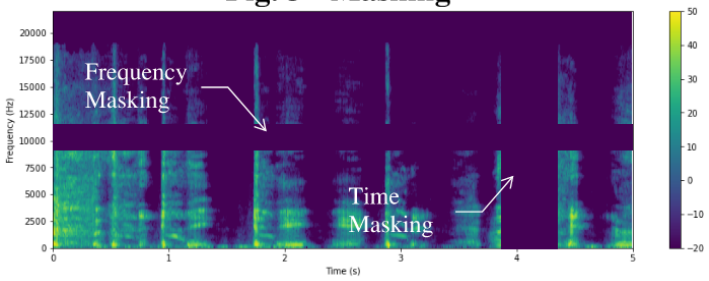
Fig. 2 - Car Honking



Data Augmentation

As is common with image classification models, augmenting audio files and their corresponding spectrograms during the training process can help create a more accurate and effective final model. There are a few common audio augmentation techniques that we experimented with and implemented during the model creation process. The first technique, known as **masking**, is similar to the image augmentation technique of cutout in that random portions of a given spectrogram are removed or zeroed out. Rather than removing random squares or sections of the spectrogram however, masking occurs at specific time intervals or frequency bands (see Fig. 3). Another set of augmentation techniques is **shifting**, which can similarly take place in both the time and frequency dimensions. Time shifting involves shifting the entire spectrogram to the left or right, cutting off the portion that falls outside of the given time-window and padding the left over space with zeroes. Pitch shifting on the other hand involves raising or lowering the frequency values of the entire audio file, thereby randomly moving the spectrogram up and down on the Y-axis. Such augmentation techniques are essential to creating a model robust to the wide varieties of variation in audio that would be captured by an edge microphone.

Fig. 3 - Masking



The Models

Training

In order to train a model to be used at the edge, we utilized the ESC-50 dataset, as previously mentioned. Fortunately, the ESC dataset is frequently used in research projects so there were reference materials available to us. However, the ESC dataset used 50 semantical classes as labels. We were only interested in one, the human non-speech sound, coughing. This made for an interesting problem, as when we exported models to use on the test dataset, sometimes the ones with higher validation accuracy were not as good at identifying coughs as those with lower accuracy.

First, we will discuss the data preprocessing and model architecture and parameters that we compared. For preprocessing, we tried two different methods, which we will refer to and method A and method B. Both methods take each audio clip and create a spectrogram, but method A uses two channels, and method B uses three channels, as well as different parameters. Snippets of the code which produces the spectrograms are included below. Please note that method B was heavily inspired by the paper "Rethinking CNN Models for Audio Classification"[2] and their code.

Method A:

```
data, sr = torchaudio.load(full_file_name)

# Resample to two channels
data = torch.cat([data, data])

# Generate a Spectrogram from the audio file
data = transforms.MelSpectrogram(sr, n_fft=1024, hop_length=None,
n_mels=64)(data)
data = transforms.AmplitudeToDB(top_db=80)(data)
```

Method B (for each channel make a spectrogram, then append to the array):

```
sampling_rate = 44100
num_channels = 3
window_sizes = [25, 50, 100]
hop_sizes = [10, 25, 50]
centre_sec = 2.5

specs = []
for i in range(num_channels):
      window_length = int(round(window_sizes[i]*sampling_rate/1000))
      hop_length = int(round(hop_sizes[i]*sampling_rate/1000))

      clip = torch.Tensor(clip)
      spec = torchaudio.transforms.MelSpectrogram(sample_rate=sampling_rate,
n_fft=4410, win_length=window_length, hop_length=hop_length,
n_mels=128)(clip)
      eps = 1e-6
```

[2] Palanisamy, Kamalesh, Dipika Singhania, and Angela Yao. "Rethinking CNN models for audio classification." *arXiv preprint arXiv:2007.11154* (2020).

```
      spec = spec.numpy()
      spec = np.log(spec+ eps)
      spec = np.asarray(torchvision.transforms.Resize((128,
250))(Image.fromarray(spec)))
      specs.append(spec)

new_entry = {}
new_entry["audio"] = clip.numpy()
new_entry["values"] = np.array(specs)
new_entry["target"] = data["target"]
values.append(new_entry)
```

In the table below, we list models we tried, the preprocessing/augmentation used, as well as the architecture and parameters.

Data Preprocessing & Augmentation	Model Architecture & Parameters	Results

Method B	Architecture: Pretrained Densenet
	`self.model = models.densenet201(pretrained=True)`
	`self.model.classifier = nn.Linear(1920, num_classes)`
	Parameters:
	`batch_size = 32`
	`num_workers = 8`
	`epochs = 70`
	`lr = 1e-4`
	`weight_decay = 1e-3`
		Best Accuracy:
		89.5%

Method B	Architecture:Pretrained Densenet
	`self.model = models.densenet201(pretrained=True)`
	`self.model.classifier = nn.Linear(1920, num_classes)`
	Parameters:
	`batch_size = 64`
	`num_workers = 8`
	`epochs = 70`
	`lr = 1e-4`
	`weight_decay = 1e-3`
		Best Accuracy:
		89.5%

Method B	Architecture: Pretrained Densenet	Best Accuracy:

	`self.model = models.densenet201(pretrained=True)`
	`self.model.classifier = nn.Linear(1920, num_classes)`
	
	Parameters:
	`batch_size = 96`
	`num_workers = 8`
	`epochs = 70`
	`lr = 1e-4`
	`weight_decay = 1e-3`
		89%

Method B	Architecture: Pretrained Resnet50
	`self.model = models.resnet50(pretrained=pretrained)`
	`self.model.fc = nn.Linear(2048, num_classes)`
	
	Parameters:
	`batch_size = 32`
	`num_workers = 8`
	`epochs = 70`
	`lr = 1e-4`
	`weight_decay = 1e-3`	

Method B	Architecture: Custom CNN
	`*first layers input size changed*`
	`See custom CNN Appendix`
	Parameters:
	`batch_size = 32`
	`num_workers = 8`
	`epochs = 70`
	`lr = 1e-4`
	`weight_decay = 1e-3`
		Best Accuracy:
		22.5

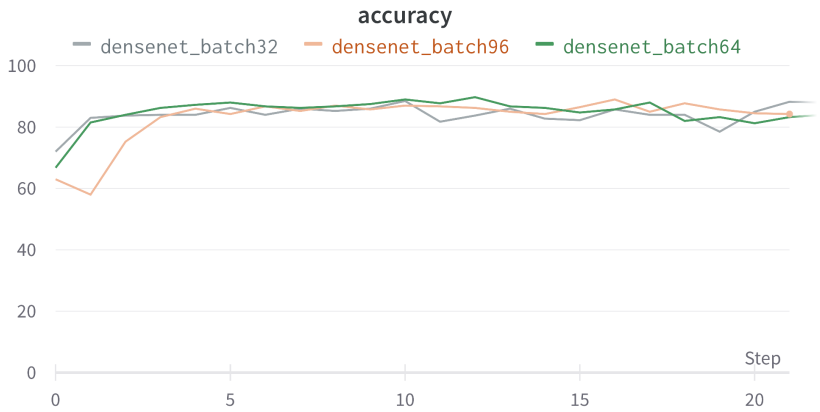
Method A	Architecture: Custom CNN
	`See custom CNN Appendix`
	Parameters:
	`batch_size = 32`
	`num_workers = 8`
	`epochs = 100`
	`lr = 0.001`	

Method A	Architecture: Pretrained Resnet34
	`self.model = resnet34(pretrained=True)`
	
	`model = self.model`
	`model.fc = nn.Linear(512, 50)`
	`model.conv1 = nn.Conv2d(1, 64,`
		Best Accuracy:
		79%

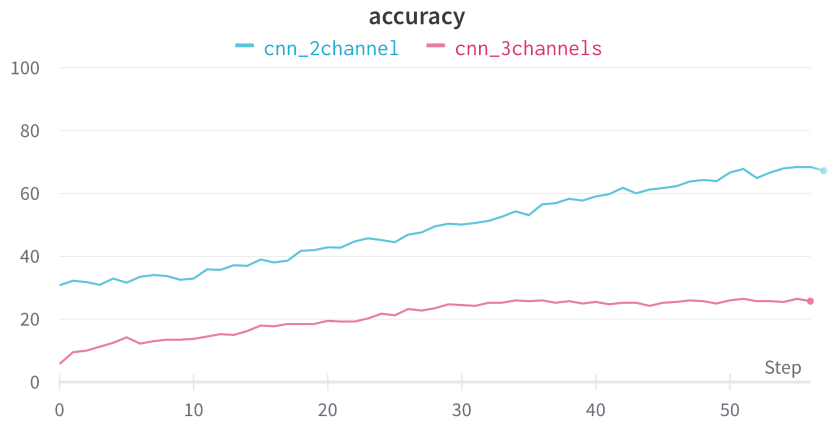
	kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False) Parameters: batch_size = 16 epochs = 100 lr = 2e^5 *with learning rate decay, see appendix*	
Method A	Architecture: Pretrained Resnet34 self.model = resnet34(pretrained=True) model = self.model model.fc = nn.Linear(512, 50) model.conv1 = nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False) Parameters: batch_size = 128 Epochs = 50 lr = 2e^5 *with learning rate decay, see appendix*	Best accuracy: 78%

Some interesting observations are as follows:

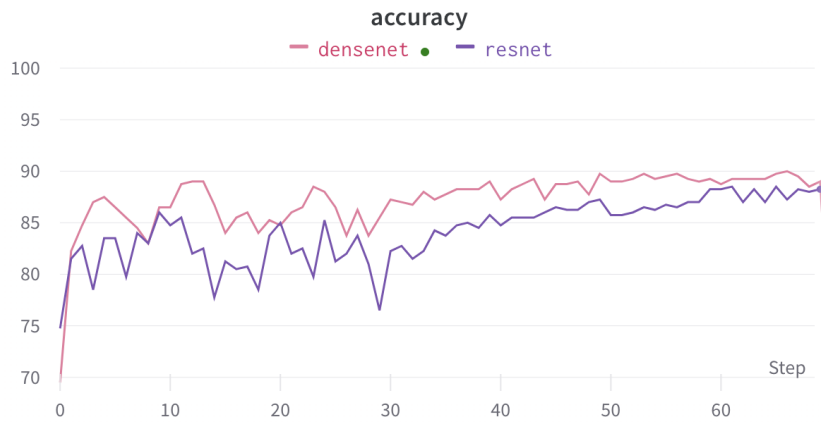
- Accuracy did not change significantly with increased batch sizes for Densenet model.



- Accuracy was significantly better when used the Method A for preprocessing than Method B.



- The Densenet model had higher accuracy, but the Resnet model provided better results on the test data (which we will discuss next).



Evaluating on Testing Data

Once we trained the model, we exported the best ones using the PyTorch code below.

```
if is_best:
        best_acc = acc
       filename = os.path.join("{}".format(checkpoint_dir), 'myModel_export.pt')
       model_cpu = model.to('cpu')
       model_scripted = torch.jit.script(model_cpu)
       model_scripted.save(filename)
```

Then, we loaded the models and used them to predict if each of the sounds in the testing dataset were coughs. All of the models exported and evaluated can be found in the “testing_exported_models.ipynb” notebook in our GitHub repository. What was most

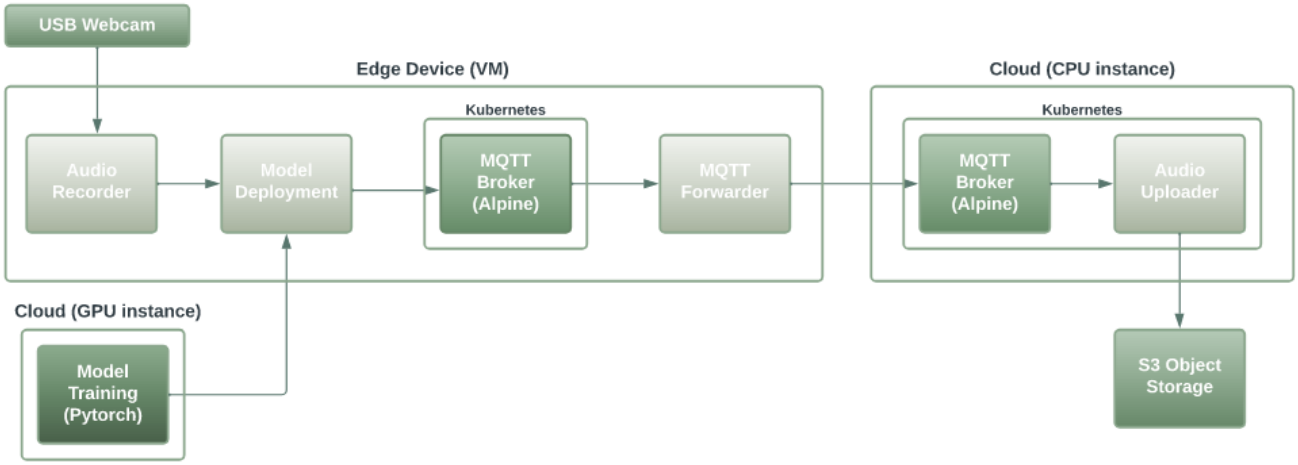
interesting was that many of the models did not label each cough as a cough, but they never mislabeled a non-cough as a cough. Furthermore, the model that identified the most coughs in the testing data (19 of the 20 coughs), was the ResNet model. This was surprising because the Densnet model had higher accuracy when training and validating on the ESC-50 dataset. The Densenet model identified 16 of the 20 coughs. This would be an interesting thing to investigate further if more time were available.

Architecture

Once we had successfully built and trained our preferred model to an acceptable level of accuracy, it was time to construct the deployment architecture that would allow for inference on the edge device and for the desired audio files to be transmitted back to the cloud for storage and eventual analysis by a medical professional. As previously mentioned, due to a lack of hardware resources, a low-powered local VM and USB webcam were used to emulate the hardware and computing/storage resources that would be available in an actual edge microphone.

The first step in the process was to create a program that would continually record audio and save it in five-second WAV audio files in a temporary staging directory. In order to run inference on these files, the model first had to be exported from the cloud GPU instance where it had been trained, which was accomplished through the use of TorchScript which saves all the weights and attributes of a model to a single .pt file which can be easily transferred and loaded on another device. A second program was created that loads the model, scans the staging directory for audio files, and then runs inferences on them one by one. If an audio file is classified as anything other than a cough, it is immediately deleted to minimize overall storage needs on the edge device. If it is classified by the model as a cough however, the file is converted to a bytearray before being passed to the MQTT message broker before also being deleted. Once all files have been read and deleted, the process repeats on all the audio files which have been added in the meantime by the recorder program.

Any files which have been passed as messages to the MQTT broker are sent out to a local forwarder program which is subscribed to the same topic, and from the forwarder the message is passed on to a broker running on a low-powered cloud instance. A final processor receives the message on the cloud, converts the byte array back to a WAV file, and uploads that file to an S3 object storage bucket on the cloud. This flow can be seen in the architecture diagram provided below.



Conclusion

Challenges

One of the first challenges we encountered was properly preprocessing the data during the inference stage on the edge device. In order to receive a prediction from a model on a single audio file, the file must first be converted into the same exact format and size that the model was trained on. Because different members of our team built models with varying types of preprocessing, each model required the development of unique testing code that had to align with that particular model. It took a substantial amount of trial and error to even begin to be able to run the testing data through some of the models that were developed.

A second challenge we faced was the limited computing power on the local VM which served as our makeshift edge device. Given the number of different processes that had to run simultaneously including the audio recorder, model deployment, kubernetes, and MQTT infrastructure, we observed a significant slowdown in the system performance that resulted in periods of lag and a staggered flow of messages being passed on to the cloud for processing and uploading. This may be the result of the complexity and size of the model, which we will discuss in the following section, or it could simply be the result of the fact that a local VM is forced to devote significant resources to the GUI, a problem which wouldn't exist if this infrastructure were to be deployed on a true edge hardware device.

Areas of Future Development

Given the time and resource constraints of this project, there were a number of improvements and developments to both our models and overall architecture that we were unable to implement. As mentioned in the previous section, the model we settled upon for deployment was quite large for a small edge device. We ran out of time to implement many of the optimization techniques such as weight pruning, clustering, and quantization that may have helped reduce the model size with minimal impact to overall performance. In turn, this may have helped address some of the slow-down and lag issues we ran into by decreasing the computing resources that need to be devoted to inference.

Additionally, we didn't have time to fully containerize and deploy all aspects of the architecture on Kubernetes on the edge device. In order to deploy this infrastructure on a true edge device this will be a necessary step as the current process requires manually starting both the recording and model deployment programs which wouldn't be possible during actual deployment.

Once this system is fully deployed and operating, an opportunity exists for further model tuning and improvement through the use of the collected data. If medical professionals

who are reviewing the collected audio files identify mislabeled data where the model labeled something as a cough that shouldn't have been, it will be useful to develop a system in which those files can be flagged and collected for use during future rounds of model training. These files will likely represent the edge cases that can be hard to collect or identify beforehand, so it will serve as a valuable set of data to enable better performance of the model moving forward.

In closing, the cough-detecting edge microphone that was envisioned and created as part of this project represents just the tip of the iceberg when it comes to the potential for edge audio devices in support of providing health care. No single object is more closely associated with the profession of doctor than the stethoscope, a device specifically designed for listening and detecting sounds to help with diagnoses. This same technology could be deployed in wearable monitors for the heart, lungs, intestines, etc. to alert both patients and medical professionals immediately when undesirable symptoms are identified. This is just the beginning of an edge technology use-case with truly life-saving potential.

Appendix

CNN Architecture:

```
class AudioClassifier(nn.Module):

    def __init__(self):
        super().__init__()
        conv_layers = []

        # First Convolution Block with Relu and Batch Norm. Use Kaiming Initialization
        self.conv1 = nn.Conv2d(2, 8, kernel_size=(5, 5), stride=(2, 2), padding=(2, 2))
        self.relu1 = nn.ReLU()
        self.bn1 = nn.BatchNorm2d(8)
        init.kaiming_normal_(self.conv1.weight, a=0.1)
        self.conv1.bias.data.zero_()
        conv_layers += [self.conv1, self.relu1, self.bn1]

        # Second Convolution Block
        self.conv2 = nn.Conv2d(8, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu2 = nn.ReLU()
        self.bn2 = nn.BatchNorm2d(16)
        init.kaiming_normal_(self.conv2.weight, a=0.1)
        self.conv2.bias.data.zero_()
        conv_layers += [self.conv2, self.relu2, self.bn2]

        # Second Convolution Block
        self.conv3 = nn.Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu3 = nn.ReLU()
        self.bn3 = nn.BatchNorm2d(32)
        init.kaiming_normal_(self.conv3.weight, a=0.1)
        self.conv3.bias.data.zero_()
        conv_layers += [self.conv3, self.relu3, self.bn3]

        # Second Convolution Block
        self.conv4 = nn.Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.relu4 = nn.ReLU()
        self.bn4 = nn.BatchNorm2d(64)
        init.kaiming_normal_(self.conv4.weight, a=0.1)
        self.conv4.bias.data.zero_()
```

```
        conv_layers += [self.conv4, self.relu4, self.bn4]

        # Linear Classifier
        self.ap = nn.AdaptiveAvgPool2d(output_size=1)
        self.lin = nn.Linear(in_features=64, out_features=50)

        # Wrap the Convolutional Blocks
        self.conv = nn.Sequential(*conv_layers)

    # Forward pass
    def forward(self, x):
        # Run the convolutional blocks
        x = self.conv(x)

        # Adaptive pool and flatten for input to linear layer
        x = self.ap(x)
        x = x.view(x.shape[0], -1)

        # Linear layer
        x = self.lin(x)

        # Final output
        return x
```

Learning rate decay code:

```
def set_learning_rate(optimizer, lr):
    '''
    set learning rate to optimizer's parameters
    '''
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr
    return optimizer

def learning_rate_decay(optimizer, epoch, learning_rate):
    '''
    set decay to learning rate every 20th epoch
    '''
    if epoch % 20 == 0:
        lr = learning_rate / (100**(epoch//20))
        opt = set_learning_rate(optimizer, lr)
        print(f'[+] Changing Learning Rate to: {lr}')
        return opt
    else:
        return optimizer
```

Additional References

1. Doshi, K. (2021, May 21). *Audio Deep Learning Made Simple: Sound Classification, Step-by-Step*. Medium. Retrieved from https://towardsdatascience.com/audio-deep-learning-made-simple-sound-classification-step-by-step-cebc936bbe5

2. Mlearnere. (2021, April 14). *Learning From Audio: Spectrograms*. Medium. Retrieved from https://towardsdatascience.com/learning-from-audio-spectrograms-37df29dba98c