Compressed U-nets for Speech Enhancement

Harry Songhurst
hs778
St Edmund's College

Ashwin Ahuja
aa2001
Gonville & Caius College

1 Introduction

Since February 2020, COVID-19 has driven a substantial change in communications. Over 45% of people have worked from home during the pandemic [1]. Online meetings, through Zoom, Google Meet and other services, have become the norm. Many people have experienced technical challenges with these services, and frustration with audio quality has become a daily issue. Most audio is captured on cheap built-in computer or phone microphones, which often exhibit hissing (static noise) and other undesirable artefacts. This noise is compounded by the natural background noise of our homes and cities. With most estimates suggesting that we will not return to full in-person work and study until mid-2021 at the earliest [2], any improvement to audio quality is welcomed.

In this project, we train multiple deep convolutional neural networks to eliminate undesirable noises from spoken-word audio. We evaluate the system on both pre-recorded and real-time speech. The former considers the specific use-case of a lecturer publishing pre-recorded lectures with bad-quality audio. We consider various categories of noise, including static noise, reverb and environmental noise. These noises are combined with clean spoken word audio to produce noisy-clean training pairs that are used to train our network. We use the LibriSpeech dataset [3] for clean spoken-word English audio and the ESC-50 dataset [4] for environmental noise.

A desirable property of any audio denoising system is the ability to denoise audio in near real-time, on consumer hardware (no GPU), with minimal latency. Hence, we consider the impact of various model compression techniques including pruning, quantization and knowledge distillation. We evaluate the models yielded through these techniques by comparing how well they denoise a given test set of noisy samples and their respective inference speeds. To assess denoising quality, we compute the RMS difference between predicted and actual clean audio and plot spectrograms to inspect the various models' strengths and weaknesses. We find, perhaps obviously, that the larger models denoise better, though they incur a penalty in inference speed. Models trained through distillation are of comparable performance but run much more quickly.

2 Related Work

Audio de-noising is a problem that has been tackled in various ways, using both deep learning and traditional approaches. Shi et al. [5] demonstrate a deep learning based approach using a CNN with skip connections (similar to ours). Defossez et al. [6] develop a system for speech enhancement on raw audio samples (without the need for spectrogram extraction). Takeuichi et al. [7] demonstrate how LSTMs can be used to achieve something similar. Many papers do not consider real-time audio de-noising, nor the problem of performing inference on consumer grade CPUs. Our work also inherits significantly from existing work on model compression, for example by Han et al. [8] and Mangalam et al. [9]

Outside of the academic world, real-time denoising has recently been implemented by Nvidia through RTX Voice [10]. This has received great critical acclaim, with reviewers extolling its virtues [11]. The details of how it works are secret, but it appears to maximise usage of CUDA cores on an Nvidia GPU. Hence, it is impossible to use for most people on cheaper computers with no GPU; a solution that works on CPUs with limited compute power would allow for much greater impact. Whilst our results are preliminary, we are convinced this is possible. RTX Voice also only denoises audio from the user microphone, whereas, we also look at the use case of denoising system-wide audio (from YouTube for example).

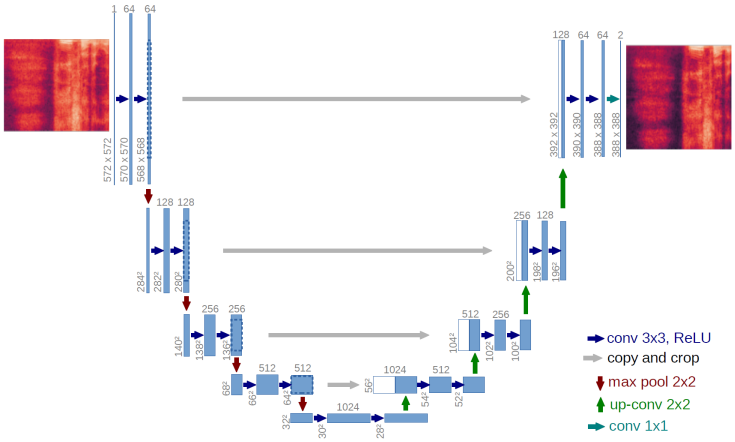

Figure 1: U-net Architecture (adapted from [12])

3 Background

3.1 Digital Signal Processing (DSP)

Digital sound files consist of a list of numbers that get translated into a voltage, which moves electrons in a coil of wire; these electrons exert a force on a magnet positioned inside the coil, moving it, and thus moving the air - giving rise to sound.

Fourier transforms are ubiquitous throughout electrical engineering and computer science. They can be used to decompose a one-dimensional signal depending on time (such as the list of numbers in our sound file), into a two-dimensional array; one dimension still codes for time, and the additional axis codes for frequencies. That is to say, if the original sound file contained a musical chord, we would expect for our Fourier transform to reveal the constituent notes of this chord.

We use the short-time-Fourier-transform (STFT) algorithm throughout our project to represent the 16,000Hz audio inputs to our network as a two-dimensional array of 128 frequencies over time (a spectrogram); this algorithm performs the Fourier transform on overlapping 'chunks' of the input, and effectively turns our audio denoising problem into 2D image denoising problem. This allows us to make better use of convolutional neural networks, which are well adapted to dealing with 2D image data. The implementation of STFT we use comes from the 'torchaudio' package, which also contains an inverse short-time-Fourier-transform, allowing us to turn the cleaned images back into audio.

3.2 U-Net

Since we frame the problem of denoising as one of reconstructing an input image from a noisy copy of itself, it is appropriate to use an encoder-decoder based architecture. The U-net [12] is one such architecture that incorporates "skip connections" which allow signal to pass from one layer in the encoding path to its corresponding layer in the decoding path, without being changed (figure 1). This is ideal because it allows most of the signal to pass from the input to the output, leaving the network to learn the structure of noise, not the (more complicated) structure of coherent spoken work audio. We experimented with multiple network depths and widths, as well as compressed variants - discussed below.

Interestingly, the U-net has a 'bottleneck' layer (figure 1), which has been studied [13] for its ability to discover abstract features in the input dataset. The idea being that using only a small number of neurons to represent the input forces the network to dedicate a few neurons to, for example, keyboard tapping noises.

3.3 Model Compression

We wanted to build a system that could enhance speech in near real-time. Since our U-net requires at least 1s of audio (to provide context to each sample) the latency of the system is effectively the time taken to run inference on a second long clip. We explored three different techniques for minimising this latency, of which one has the side effect of reducing the size of the model on disk (allowing for cheaper distribution). We look at the impact of these methods on the quality of speech enhancement and inference speed.

3.3.1 Pruning

Pruning is the process of systematically removing either parameters, or entire neurons, from a network. We assume that our network is over-parameterised, and that it contains parameters or neurons that do not affect our output. By pruning, we obtain a smaller network with performance comparable to the larger, un-pruned network. Pruning was introduced in the late 1980s [14] but has seen a resurgence in usage as deep learning has become more popular. Briefly, our use of pruning proceeds as follows:

1. Begin with initial network.
2. Evaluate the importance of each of the filters in the network, finding the least important filter.
3. Eliminate this filter.
4. Tune the network (train for some number of epochs).
5. Repeat from (2) for a certain number of iterations.
6. Finish with a pruned network.

To prune effectively, we must be able to measure the importance of individual parameters in the network. One way to do this is simply take parameters' magnitude as their importance. However, we chose to use a more novel approach for pruning devised by Molchanov et al. [15] for convolutional neural networks, which allows us to prune away entire filters. First and second-order Taylor expansions are used to approximate each filter's contribution/importance. Molchanov et al. show this has a significant improvement over other existing methods.

Pruning can be carried out either during training or after training. We expect better performance from pruning during training as the model adapts to the new architecture. Therefore, we chose to prune in this manner.

3.3.2 Weight Quantization

In a deep-learning context, 'quantization' refers to changing the bit-width of the tensors used to store the parameters of a model [16]. A reduction in the number of bits used for each weight can make inference speeds quicker, and reduce the size of our model on disk. PyTorch allows us to approximate floating points (FP32) weights with integers (INT8), in turn allowing us to take advantage of both software and hardware optimizations that exist for integers. Quantization is usually performed after training - we train a model to completion with FP32 weights, then convert these weights to INT8. PyTorch also supports 'quantization aware' training, which we keep track of, and adapt to, errors that propagate because of quantization. Because of this, we expect better performance - though it comes at the cost of additional compute requirements, and extra engineering time (we must make our model compatible with quantization aware training) - for these reasons, we only explored post-training quantization.

3.3.3 Knowledge Distillation

Knowledge distillation, originally introduced by Hinton et al. [17], is a technique for reducing the size of a model, thus speeding up inference. The paper describes how a 'teacher' model, with a large size and perhaps slow inference speed, can be used to train a smaller 'student' model. We initially train the teacher to completion, then run our training for the student on the same dataset, however, with the labels replaced by the output of the teacher network, instead of the 'actual' labels. It is theorised that this works because the teacher predictions contain 'dark knowledge'; in the case of a hand-written digit classifier, a 7 might look a lot like a 1, and the teacher confers this knowledge to the student by outputting a distribution that is perhaps 70% 7, 30% 1.

Mangalam et al. [9] showed that this technique can be used for compressing U-nets - our architecture of choice. We compared distillation to simply training a lower depth model and found it offered statistically significant improvements. It is important to note that training a knowledge distillation model effectively requires training two separate models and therefore takes a long time.

We defined our 'teacher' model as a U-net with a channel depth of 64, while the 'student' model had a channel depth of 32. This yielded models of approximately 25% the size ($\approx$13MB vs $\approx$53MB).

4 Methodology

4.1 Data Assembly

We first considered finding datasets that had both noisy and clean audio of the same voices. However, this offered little flexibility over specifying the types of noises we wanted to remove/clean-up. Therefore, we instead decided to compile a small dataset of realistic environmental noises from ESC-50 [4] and wrote various functions for generating synthetic noise (static and reverb). Clean audio came from the LibriSpeech - a dataset of 1000 hours (5GB) of audio-book quality spoken-word english [3]. All samples in this dataset have a sample rate of 16KHz, which is double the minimum frequency for capturing all frequencies of human voice (8KHz).

Each clean training sample had a random selection of the following noises added to it at each training iteration:

- **Static noise:** This was generated by simply adding some Gaussian noise to clips.
- **Reverb:** In order to generate reverb, we used the method described by Defossez et al. [6]. Here, a succession of attenuated echos are added to the input signal itself. These echos are added with reducing amplitude. The delay between echos is defined by a constant that mimics the radius of the room where the audio was recorded - we chose this to be a random number simulating a room between 10 and 30 meters wide.
- **Environmental Noise:** We utilised the ESC-50 dataset which provides 50 different noises. We hand-selected the following background noises as the most likely: keyboard taps, coughing, clock-ticks, clicks and wind. For each of the categories, we listened to some samples and determined a useful amplitude scalar that ensured that the noise was noticeable, but not overpowering.

We trained various models on three different noise configurations to assess whether certain noises resulted in worse overall model performance. First, we train on all types of noise, each selected with a random probability. Second, we train on all types of noise, however, we sample each noise with weighted probabilities (for example, keyboard noises were selected more). Third, we train with all noises except reverb - since we found that reverb drastically decreased performance on other types of noise, and made the model too keen to remove anything and everything.

4.2 Creating Models and Training

We submit 14 models, all of which were trained and evaluated in their own self-contained iPython Notebooks using PyTorch. We made use of Google Colab for this purpose, since they provide free access to powerful GPUs. Minimal code required to train a denoising model can be found here along with demonstration audio clips that were denoised by the trained model. Our repository [18], of course, contains many more notebooks with more involved code.

Out initial U-net architecture was programmed in accordance with figure 1. Additionally, we defined a smaller model where one of the down-sampling and one of the up-sampling layers were removed. For pruning, we modified existing U-net pruning code [19]. All models were run for 5 epochs over the entire 5GB LibriSpeech dataset. It took around 8 hours to train a single model - we did not evaluate or monitor the training time precisely since this did not impact the usability of the final system. Quantization was carried out after training of a model and was completed using the inbuilt Dynamic Quantization method that PyTorch provides. However, it is important to note that this method's implementation is rather limited and it only quantizes linear layers, leaving Convolutional layers (most of the layers in the U-net) entirely alone [20].

All told, we experimented with the following architecture and training configurations. Each model was trained with three different configurations of noisy data, as previously discussed in 4.1.

- Normal Model (all u-net layers)
- Smaller Model (two fewer layers)
- Normal Model with Pruning
- Smaller Model with Pruning
- Normal Model with Quantization
- Smaller Model with Quantization
- Normal Model with Knowledge Distillation - therefore outputting a model with half the channel depth (32 channels vs 64 channels of all other models)

- Normal Model with Pruning and Quantization
- Smaller Model with Pruning and Quantization
- Normal Model with Quantization and Knowledge Distillation

The trained versions (and training notebooks) of each of these models is available in the repository.

4.3 Model Distribution

We looked into multiple methods of distributing our final model. Ideally, we would have delivered a system that allowed a user to simply click a button, and have all system-wide spoken word audio piped through our model before being sent to the speakers. We looked into doing this using a Chrome extension, and Apple 'AudioUnits'. Both proved too difficult to get working reliably; we hope to come back to this in the future (6).

As a simplified method, we produced a Python application which maintains a circular buffer of the last 1s of audio from the user microphone, and passes it through our model before playing it back to through the speakers. This runs in near real-time, with a $\approx$200ms delay, and successfully denoises audio. We envision this being used with an application such as 'SoundFlower' or 'Blackhole', which allows you to pass audio from a specified application (e.g. VLC media player) through another application (our python script) before playback occurs through the speakers.

We also created a Flask application which downloads a video from a given YouTube link using 'youtube-dl', splits out the audio using FFMPEG, processes this audio through our model, then joins everything back together before allowing the user to download the denoised video. A demonstration of this is available in the repository, alongside instructions on how to run it, however, it runs too slowly without a powerful GPU to be of much use. Ironically, it is better to denoise in near-real-time with a slight delay at the start. This way you get denoised audio on-demand, whereas with this application, you would need to wait 5 minutes to denoise a 5 minute video, which feels much longer.

5 Evaluation

We evaluated all trained models with respect to 'model efficacy' (how well they actually denoised audio), and inference speed. In order to find the efficacy of the model, we run inference on LibriSpeech's test dataset with noise added as before. We determine the Root Mean Square difference in the spectrograms of the original clean clip and the denoised version produced by the model. The average of this RMS difference is found over the test dataset. To find the time taken for inference, we use the timeit function provided by iPython. Here, inference is carried out thirty times in three rounds. The minimum average time for a round of inference is taken and compared between models. All inference times quoted are on an Nvidia K80.

We would have liked to have trained each model for 200 epochs, as is done in [6], however we lacked the requisite computational resources. We observed that after 5 epochs (10000 iterations, taking $\approx$8 hours on an Nvidia K80) each model reached a fairly respectable performance (both quantitatively, and subjectively). Figure 2 plots representative training losses for each (larger) model before any compression.

For each of these model, and smaller models (with one convolutional, and one transpose-convolutional layer removed), we compute the root mean squared (RMS) error between predicted magnitude spectrograms and target magnitude spectrograms, over the same test set. Table 1 shows these results.

	RMS Error			**Inference Speed / ms**		
	Weighted Probs	**Equal Probs**	**No Reverb**	**Weighted Probs**	**Equal Probs**	**No Reverb**
Normal Model	2.85	3.19	2.30	241	243	243
Smaller Model	2.92	3.46	2.97	211	213	203

Table 1: Table of mean RMS and inference speed for normal and smaller model for different noise profiles

These results show how the version with no reverb performs best, followed by the version with weighted noise, followed by equal probabilities. This makes sense as each adds additional complexity. Both adding reverb and setting the probability of all noises to be equal increases the entropy of the

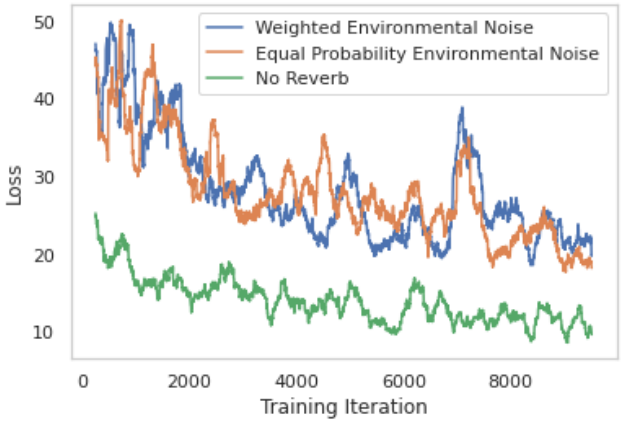

Figure 2: Losses for normal model during training

system. This explains the reduced performances of these systems. We also note that the smaller model performs statistically significantly less well. Statistical significance is verified with a paired t test. This uses a null hypothesis that the two models are the same and we use an alpha value of 0.1.

The latency results are as expected. They demonstrate that irrelevant of the noise profile used, the inference time is based only on the architecture of the model used. The smaller model runs significantly faster (approximately 12%) than the larger model.

5.1 Model Compression

We then consider the impact of pruning, quantization and knowledge distillation. Table 2 and Figure 3 shows these results.

	Compression	RMS Error			Inference Speed / ms
		Weighted Probs	Equal Probs	No Reverb	
Normal Model	**None**	2.85	3.19	2.30	242
	Pruning	2.93	2.84	2.78	225
	Quantization	3.35	3.20	2.40	233
	Knowledge Distillation	3.33	3.11	2.95	95.6
	Pruning and Quantization	3.06	3.12	3.45	233
	Quantization and Knowledge Distillation	3.40	2.99	3.24	95.8
Smaller Model	**None**	3.11	3.46	2.97	212
	Pruning	2.96	3.05	3.39	201
	Quantization	3.37	3.50	3.27	203
	Pruning and Quantization	2.90	3.36	3.60	203

Table 2: Mean RMS and Latency for normal and smaller model for different noise profiles with various model compression techniques.

We expect each of the model compression methods to reduce the denoising efficacy, but also improve the inference speed. With reference to the inference speeds we measured, this appears to be generally true - though the impact of various techniques is marginal. For pruning, we show a drop of around 10%. This perhaps shows that the u-net architecture has a high relevance density, with all the filters being reasonably important. Quantization exhibits drops of around 5%, but we find that this difference is not statistically significant and warrants more testing. We can reasonably connect this to the limits of PyTorch's implementation of dynamic quantization which only quantizes linear layers. We expect that training aware quantization (working on all layers) would have a much more significant impact. Knowledge distillation has a very significant impact, dropping latency below a tenth of a second.

Looking at the denoising efficacy, pruning seems to have a slightly negative effect, increasing the RMS for both the weighted noise probabilities and the no reverb variants. It, however, seems to improve the equal probabilities variant. Quantization universally increases the RMS error for all models, including

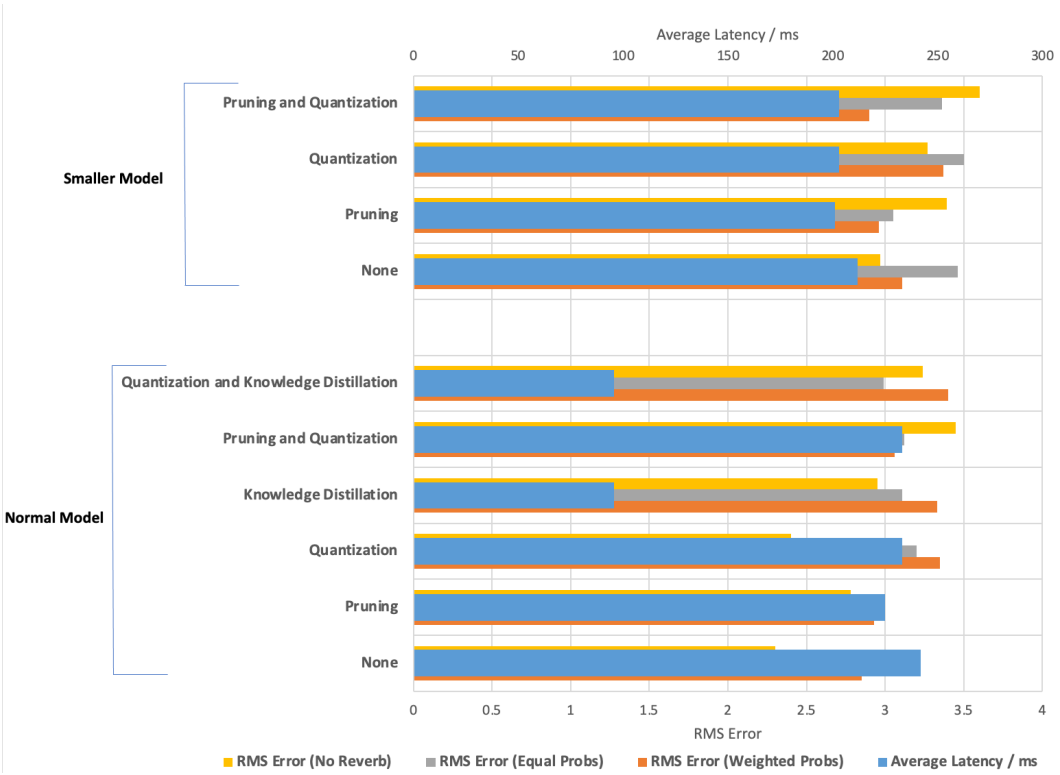

Figure 3: Bar Chart of mean RMS and Inference Speed.

pruned ones. This indicates that the current use of quantization offers minimal latency improvements with a significant conversion overhead. Finally, knowledge distillation shows similar results to pruning, with poor effects for weights and no reverb, but a slight improvement for equal probabilities. This warrants more testing.

We can compare the impact of different compression models on noise by looking at input and output spectrograms from both white and environmental noise (Figure 4) and reverb (Figure 5).

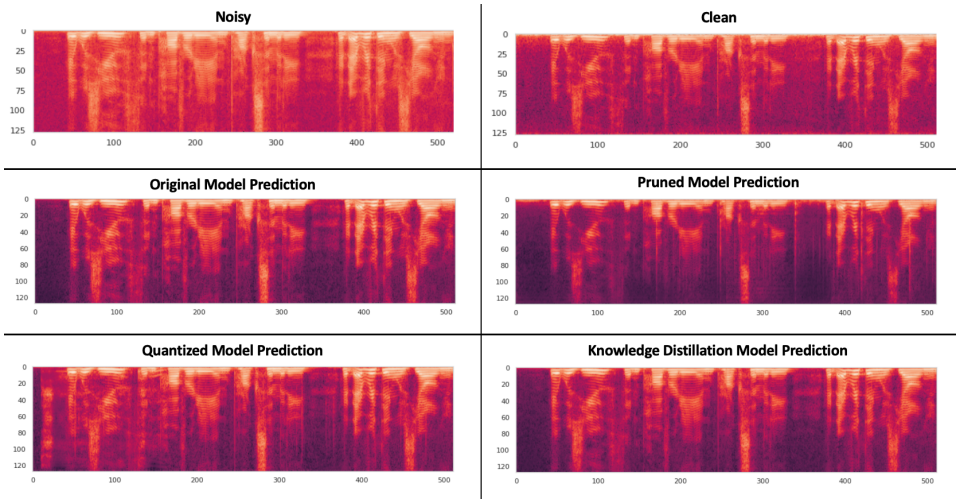

Figure 4: Spectrogram of u-net predictions on a single test audio clip from different compressed models

Figure 4 shows how all methods effectively retrieve the peaks of the cleaned target. In fact, they all seem to partially overshoot, with the background noise even less than in the initial clean spectrogram. The pruned version appears to lose some of the higher frequency items with gaps in the peaks. Finally,

with knowledge distillation, we appear to get very similar results as the original model. Therefore, it indicates this method has worked well.

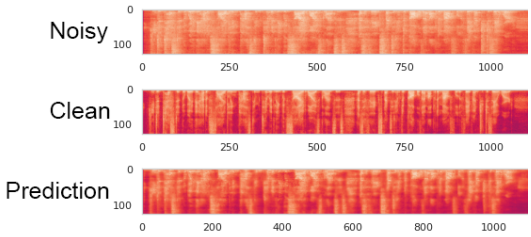

Figure 5: Spectrogram of sample with reverb, target, and prediction from original model.

5.2 Real Time Evaluation

Using the knowledge-distilled version of the model, we evaluated performance on an 8 core Intel laptop i9 CPU. We observed that the latency of this model was around 250ms. While this is still rather significant, and could pose some issues when audio is synced to video, from testing, this appears reasonably usable. It is worth noting that modern browsers support AV-sync requests from the operating system; meaning that a Youtube video could be 'told' to stall its video playback by 250ms.

6 Future Work

Apple provide an ecosystem of audio processing utilities and integrate to the macOS operating system. We attempted to compile our best model into an "AudioUnit v3" plugin, which is essentially a self contained program that inserts itself in the OS audio processing graph between an audio source (the system) and an output (speakers). This was much harder than we anticipated; AudioUnits are set up to process single bytes of audio in series, using low level C++ routines. We would have had to rewrite all of the DSP and audio-processing utilities in C++, figure out a way to maintain a buffer of the last 1s of audio, and figure out a way to quickly and continuously pass that 1s of audio through our model. For what it is worth, we may come back to this at a later date, as being able to simply click a button and have all system audio get cleaned up would be very useful, and we are convinced it is possible, since our best model only takes 100ms to process 1s of audio. This processing time may be even further enhanced by making use of Apple's native Core ML framework.

Another important item to consider is training aware static quantization. This would allow us to more comprehensively review the impact of quantization and perhaps including it as part of usable models. The dynamic quantization that we used here had little impact on the latency, whilst significantly increasing the RMS error - this is likely due to the fact that PyTorch quantization does not work for convolutional layers out of the box. [20]

Finally, we would like to complete some more realistic testing. In particular, we expect the results from simulated noise testing to be reasonably representative of the real world. In particular, preliminary testing using the Python program showed that the denoising worked reasonably effectively. However, it would be useful to create a real-world noisy testset and run our model on this. Establishing a ground truth for this would be challenging however, and perhaps require some thought. It would also be useful to compare the system we've produced with commercial GPU-based denoising systems, such as Krisp.ai [21] and RTX Voice [10].

7 Conclusion

We trained many different permutations of a U-net on the task of enhancing spoken word audio. Our results, both quantitative and subjective indicate that our method works well for removing static noise, keyboard tapping and reverb. We investigate various model compression techniques, and find that knowledge distillation can be used to greatly improve inference speed, at little sacrifice to denoising efficacy. We distribute this model through both a Python real-time application for denoising microphone input, and a Flask application for denoising Youtube videos.

References

[1] O. of National Statistics, "Coronavirus and homeworking in the UK - Office for National Statistics." https://www.ons.gov.uk/employmentandlabourmarket/peopleinwork/employmentandemployeetypes/bulletins/coronavirusandhomeworkingintheuk/april2020, 2020. (Accessed on 12/30/2020).

[2] McKinsey, "When will the COVID-19 pandemic end?." https://www.mckinsey.com/industries/healthcare-systems-and-services/our-insights/when-will-the-covid-19-pandemic-end#, 2020. (Accessed on 12/30/2020).

[3] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an ASR corpus based on public domain audio books," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pp. 5206–5210, IEEE, 2015.

[4] K. J. Piczak, "ESC: Dataset for environmental sound classification," in *Proceedings of the 23rd ACM international conference on Multimedia*, pp. 1015–1018, 2015.

[5] Y. Shi, W. Rong, and N. Zheng, "Speech Enhancement using Convolutional Neural Network with Skip Connections," in *2018 11th International Symposium on Chinese Spoken Language Processing (ISCSLP)*, pp. 6–10, Nov. 2018.

[6] A. Defossez, G. Synnaeve, and Y. Adi, "Real Time Speech Enhancement in the Waveform Domain," *arXiv:2006.12847 [cs, eess, stat]*, Sept. 2020. arXiv: 2006.12847.

[7] D. Takeuchi, K. Yatabe, Y. Koizumi, Y. Oikawa, and N. Harada, "Real-time speech enhancement using equilibriated RNN," *arXiv:2002.05843 [cs, eess]*, Feb. 2020. arXiv: 2002.05843.

[8] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[9] K. Mangalam and M. Salzamann, "On Compressing U-net Using Knowledge Distillation," *arXiv:1812.00249 [cs, stat]*, Dec. 2018. arXiv: 1812.00249.

[10] Nvidia, "NVIDIA RTX Voice: Setup Guide." https://www.nvidia.com/en-us/geforce/guides/nvidia-rtx-voice-setup-guide/#RTX-Voice, 2020. (Accessed on 12/30/2020).

[11] "Nvidia's RTX Voice app was great, and its Broadcast successor is now available - The Verge." https://www.theverge.com/2020/9/17/21444508/nvidia-broadcast-download-rtx-voice-noise-app. (Accessed on 01/09/2021).

[12] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *arXiv:1505.04597 [cs]*, May 2015. arXiv: 1505.04597.

[13] N. Tishby and N. Zaslavsky, "Deep Learning and the Information Bottleneck Principle," *arXiv:1503.02406 [cs]*, Mar. 2015. arXiv: 1503.02406.

[14] S. A. Janowsky, "Pruning versus clipping in neural networks," *Physical Review A*, vol. 39, no. 12, p. 6600, 1989.

[15] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz, "Importance estimation for neural network pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 11264–11272, 2019.

[16] PyTorch, "Quantization — PyTorch 1.7.0 documentation," 2020.

[17] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," *arXiv:1503.02531 [cs, stat]*, Mar. 2015. arXiv: 1503.02531.

[18] "GitHub Project Code Repository." https://github.com/indrasweb/chvoice. (Accessed on 01/13/2021).

[19] PyTorch, "Leekwanmeng/pytorch-unet-pruning." https://github.com/Leekwanmeng/Pytorch-Unet-Pruning. (Accessed on 12/30/2020).

[20] P. Forums, "Cannot quantize nn.conv2d with dynamic quantization - quantization - pytorch forums." https://discuss.pytorch.org/t/cannot-quantize-nn-conv2d-with-dynamic-quantization/66722. (Accessed on 12/30/2020).

[21] K. AI, "Krisp noise cancelling app." https://krisp.ai/. (Accessed on 12/30/2020).