Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра математики факультету інформатики

Підвищення частоти дискретизації звуку за допомогою методів збільшення роздільної здатності

Курсова робота
за спеціальністю „Прикладна математика ” 113

Керівник курсової роботи
к.ф.-м.н., ст. в. Швай Н.О.

_____
(підпис)

“____” _____ 2021 р.

Виконав
студент 1 курсу
факультету інформатики
Процик О.І.

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри математики,
проф., д.ф.-м.н.
_____Б. В. Олійник
(підпис)
„____”_____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Процику О. І. факультету інформатики 1-го курсу МП
ТЕМА Підвищення частоти дискретизації звуку за допомогою методів збільшення роздільної здатності
Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Календарний план
3. Вступ
4. РОЗДІЛ 1: Загальна інформація про задачу та нейронні мережі
5. РОЗДІЛ 2: Аналіз нейронної мережі та порівняння з іншими методами
6. Висновки
7. Список використаних джерел

Дата видачі „___” _____ 2021 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Календарний план виконання роботи:

№	Coursework writing stages	Date completed
1.	Started researching audio super-resolution	28.11.2020
2.	First working prototype	10.01.2021
3.	Received coursework topic	02.12.2020
4.	Completed training of the model	13.03.2021
5.	Completed writing underlying theory	07.05.2021
6.	Completed coursework draft	09.05.2021
7.	Completed final coursework draft	10.05.2019

Студенту Процику О.І.

Керівник Швай Н.О.

“_____”_____

Table of Contents

Introduction

Each day, quintillion bytes of information is being transferred through the internet. According to a study, people around the world are watching over 4 million videos simultaneously, every minute of the day. That doesn't include music streaming, voice messages or other media on the internet. All this would amount to a large chunk of data being transferred, consuming vast amounts of data, be it as storage in the cloud or on the user devices. One of the largest sources of transferred information types is media, or rather audio content. In order to optimize the amount of data being transferred, compression methods take place, or a more strict methodology, audio downsampling, which introduce noticeable fidelity degradation. Audio downsampling is the process of lowering audio sample rate, which is the basis of any audio sequence, representing the amount of observed frequencies every second. On the contrary is audio upsampling, which is the process of elevating the sample rate.

Audio upsampling methods nowadays rely on purely signal processing and statistical algorithms, which are good to a certain degree. The best algorithms still struggle to retain the fidelity of the source audio, even when upsampling by a factor of 2. The generative modeling of audio signals is a fundamental problem of signal processing. Looking at the rise of generative modeling using Neural Networks, it's an seems as an obvious extension to the current methods of audio upsampling. To formulate the task, audio super-resolution, neural upsampling, or bandwidth extension is the task of generating high sampling rate audio signals with full frequency bandwidth from low sampling rate signals.

The aim of this research is to analyze the current methods of audio upsampling and the analysis of the capabilities of Neural Networks as generative models in the audio upsampling domain. The task of this research is to conduct thorough

experiments on the fidelity and execution time of current signal processing methods and to create and train a Neural Network, which would have a competitive comparison to original methods, while producing higher fidelity audio.

This work is split into two sections. The first section aims at introducing the theoretical knowledge of basic components behind generative Neural Networks, how they are trained and a deeper insight into signal processing, including the ways we can utilize it to build better generative models. The second section aims at building a Neural Network, training it, analyzing the current methods and comparing them experimentally, introducing metrics at comparing them.

1.Theoretical knowledge of Neural Networks and signal processing

1.1.Basic audio processing concepts

An audio signal is the representation of sound, typically indicating the change in electric voltage in the case of analog signal, or a series of binary numbers for a digital signal. It can be expressed as a continuous waveform on the axis of time and axis of frequency recorded. The frequency axis typically ranges from 20 to 20,000 Hz, which is close to the upper bound of human hearing. This leads to it being unnecessary to record higher frequencies, if the aim is the media for human consumption.

Audio signal processing is used to convert between analog and digital formats, to manipulate the frequency ranges, remove unwanted noise, or even add audio effects. The first necessary component of audio capture is an analog to digital converter, which takes in the recording of electric signals and transforms it into a series of binary representations. Converting it into a digital representation uses a fixed sample rate, which is the amount of frequency measurements per each second. It is natural that the higher the sample rate, the more precisely we can convey audio with minimal degradation. The performance of analog to digital converter (ADC) is defined by signal-to-noise ratio (SNR), and the bandwidth is characterized by the sample rate. The output of the ADC is a digital signal, the frequencies of which are usually represented on a scale from -1 to 1. The discretization of frequencies is usually done with a 32 bit number in speech, there is a total of 32767 different possible frequencies. It is common to set a higher bit-rate for content, that requires greater quality, like music. Although 32 bits is enough to convey speech.

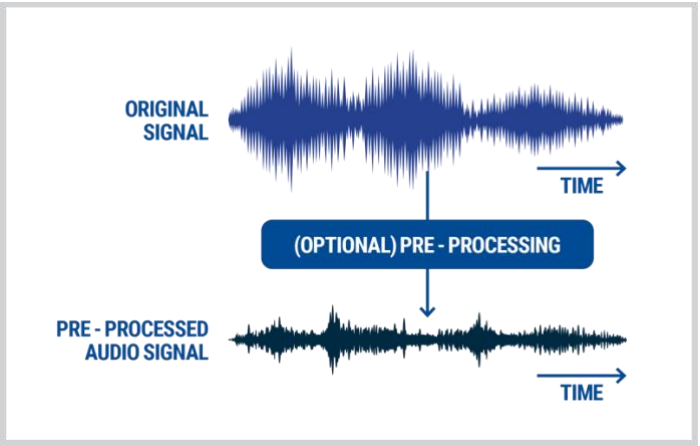


Figure 1.1 – Analog to digital signal process

Any pre-processing to the audio can be done with the binary representation, in order to improve the quality, reduce SNR, remove unwanted noise or bias brought in when recording the analog signal. This process is visualized in Figure 1.1.

1.2. Short-time Fourier transform

A spectrogram can be defined as a visual representation of the sound. It shows the frequencies that make up the sound and how they change over time. It can be generated by applying Short-time Fourier transfer to a digital signal. The Short-time Fourier transform is a transformation, that is used to determine the sinusoidal frequency and phase deltas over time. We can formulate it as:

$$X(m,k) = \sum_{n=0}^{N-1} x[n+mH]\omega[n]e^{-\frac{2\pi ikn}{N}}$$

Where

$x: [0: M-1] \coloneqq \{0,1,\dots M-1\} \rightarrow \mathbb{R}$ real valued discrete time audio of length M received by sampling an audio signal at a constant sampling rate R.

$\omega: [0: N-1] \rightarrow \mathbb{R}$ window function of length $N \in \mathbb{N}$.

$H \in \mathbb{N}$ is the hop size, or by what amount we shift the window function.

$m \in [0:M]\ where\ M := \left\lfloor \frac{L-N}{H} \right\rfloor$ is the maximal window index.

$k \in [0:K]\ where\ K = \frac{N}{2}$ frequency index corresponding to Nyquist frequency.

The complex representation *X(m,k)* stands for the k[th] Fourier coefficient of the m[th] frame. As a result, we for every frame we get a vector of length K. Thus we can efficiently transform an audio signal $x \in \mathbb{R}^n$ into $s \in \mathbb{R}^{K \times M}$, a 2-D representation of sound which we can further analyze.

The main concept of it being short-timed, is the usage of a cropped segment of the input signal, instead of using it all at once. This is achieved by using a window function, which is non-zero only a certain period of time. Then the signal is multiplied by the window function and FFT is applied on that small segment, to get a frame of the spectrogram. This is the basic explanation behind the way STFT is computed.

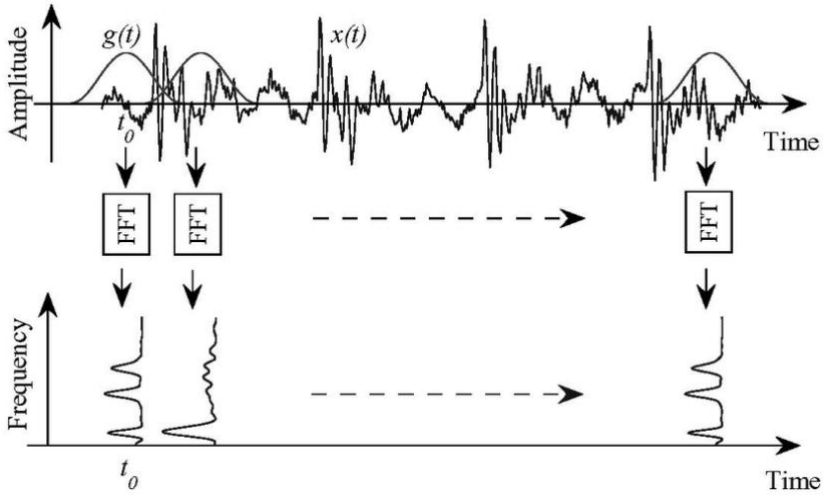


Figure 1.2 - Visualization of STFT on audio.

On Figure 1.2 the whole process of STFT is visualized. The final components that are needed are a few user-set parameters. They are window

function, window length, hop size, and the number of STFT bins. The window function is the function, that we apply over the signal. There are several common window functions such as rectangular, triangular, Hann, Bartlett, Hamming windows. The window length is the length of the non-zero part of the window function. We can effectively call that the number of samples that are taken into the window. The hop size is the size, in samples, of the distance between windows. The number of FFT bins is effectively the number of bins we are decomposing the signal into with FFT. For voice related activities, 1024 is sufficient.

By setting all these parameters, we can easily calculate the shape of the output spectrogram, given the size of the input signal. It can be calculated with the following formula.

$$S \in \mathbb{R}^{d_{spectr} \times \left\lceil \frac{signal\ length}{hop\ size} \right\rceil}$$

In this formula, S is the resulting spectrogram, d_{spectr} is the number of FFT bins. In order to localize the digital signal in time, it is necessary to use a windowing function, which would fade out on its ends. This is done to remove unnatural discontinuities in the resulting spectrogram. The window function which will be used further on is the Hann window function.

$$\omega[n] = 0.5\left(1 - \cos\left(\frac{2\pi n}{N}\right)\right), \qquad 0 \leq n \leq N$$

Where the window length is $L = N + 1$.

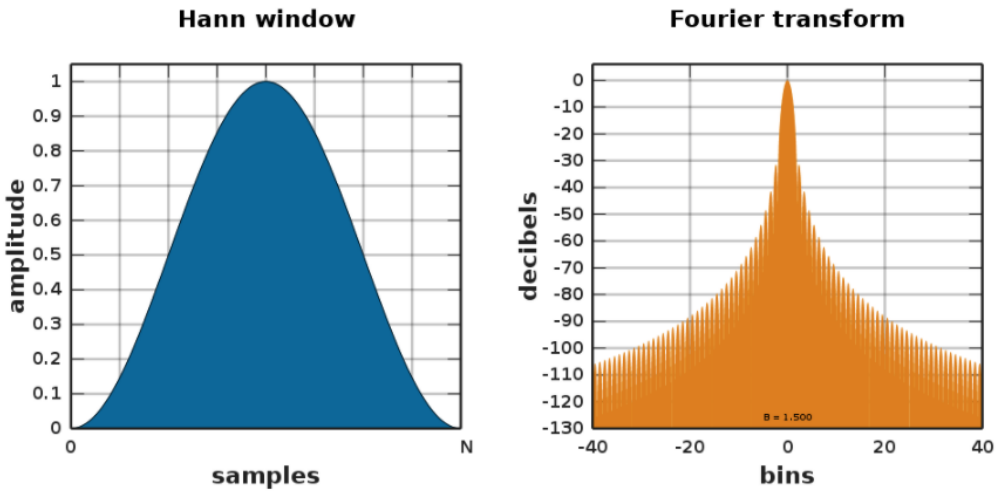


Figure 1.3 - Hann window visualization

In Figure 1.3 the Hann window function is visualized. This window function is applied to the segments that are cropped from the full digital audio signal. The window has the size of window length, which is set by the user, and the amount of samples that it slides over through the signal is to compute the next frame is determined by the hop size.

1.4. Task formulation

The generative modeling of audio signals is a fundamental problem at the intersection of signal processing and machine learning. With the recent advances in machine learning in speech recognition, speech synthesis, and other areas, it is possible to have enough computational and parametric capacity to operate directly on the audio signal.

This paper is focused on an audio generation task, audio super-resolution, neural upsampling, or bandwidth extension, which is the task of reconstructing a high-quality audio signal from a lower quality signal, with fewer samples, lower sample rate.

1.5. Perceptron

Perceptron is a simple one layer network. A network with multiple layers is called a Neural Network. A perceptron is usually used for binary classification or regression.

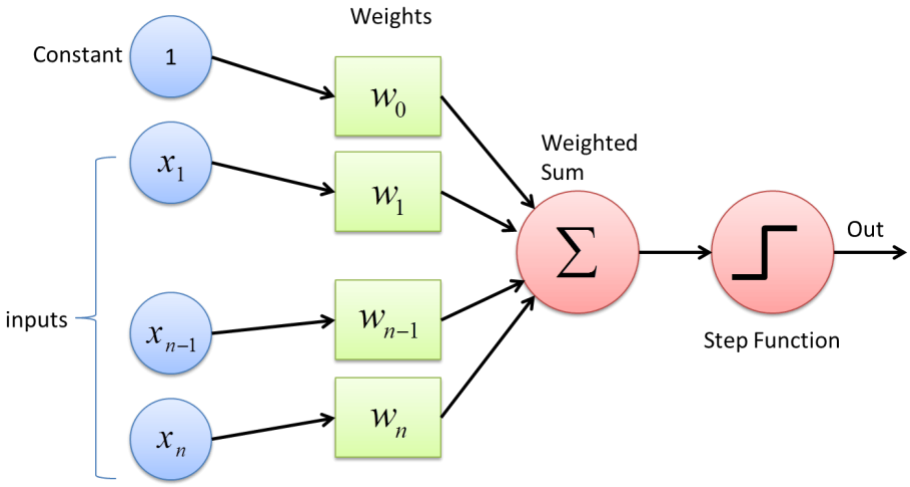


Figure 1.4 - perceptron visualization

In the case of regression, we can interpret a perceptron as a linear regression. This is a visualization of a Rosenblatt's thresholded perceptron. It tries to mimick the basic building block of the brain – a biological neuron. It takes an input vector ***x*** and outputs a number ***y***. We can formulate it by the following formula.

$$z = \sum w_i x_i = w^T x$$

We then pass ***z*** through an activation function, specifically the step function, which is defined as:

$$g(z) = \begin{cases} 1 \; if \;\; z \geq 0 \\ -1 \; otherwise \end{cases}$$

It is necessary that this function is differentiable. The reason for that is the way a neural network learns, with the process called backpropagation . We add in a constant, usually 1, as the first entry of the input vector ***x*** and is then expressed as

the bias of the network. In the case of a perceptron, the bias can be interpreted as moving the line or plane away from the origin.

1.6. Multi-Layer Perceptron

A perceptron with multiple hidden layers and activation functions is called a multi-layer perceptron, or is a basic neural network. A hidden layer is the vector of parameters, as in the example of the perceptron. Multiple consecutive layers provide a higher complexity, and the network can be visualized as in Figure 1.5.

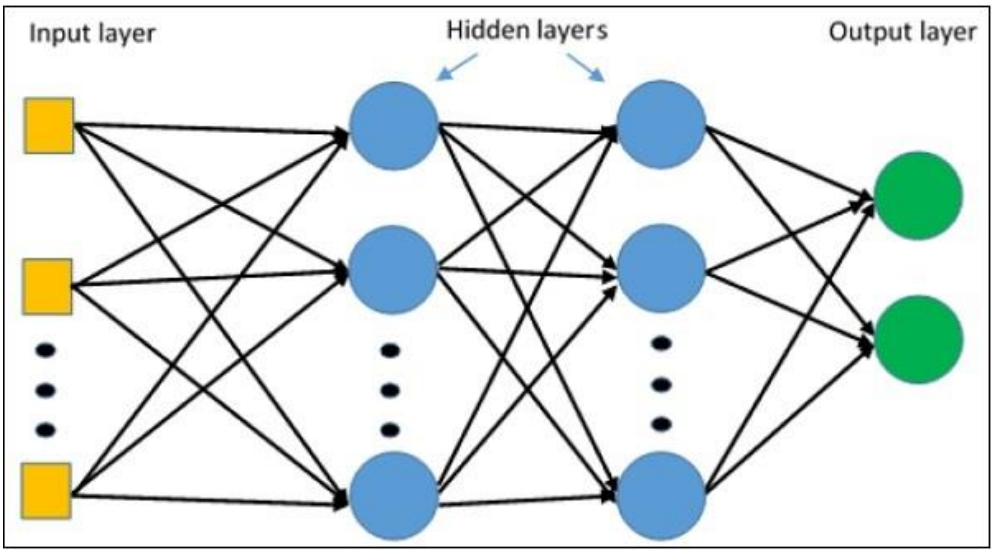
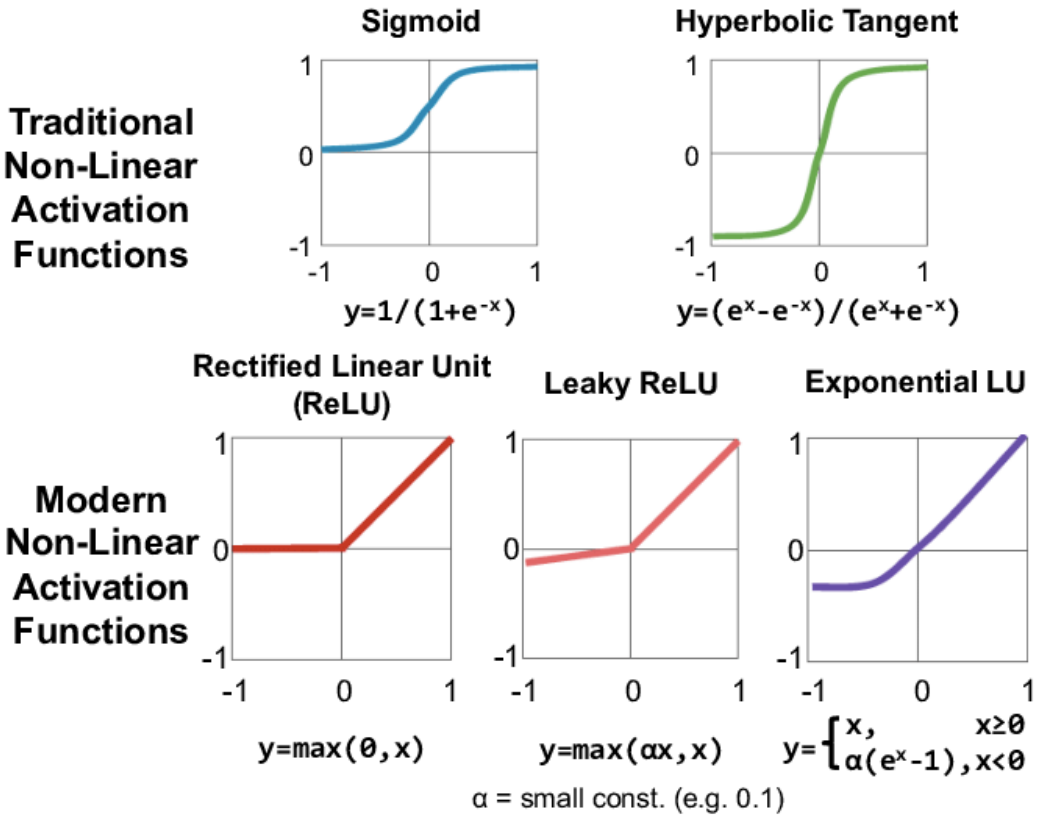


Figure 1.5 - Multi-Layer perceptron

1.7. Activation function

The outputs of each layer of a Neural Network is plainly a weighted sum of its inputs, which lies on the range of $(-\infty, +\infty)$, which would make it hard, if not

impossible for the network to learn efficiently. The other downside is that a weighted sum is just a linear transformation, which reduces computational capacity and the resulting feature space that can be constructed by the network. In order to limit the range of possible values to suit a specific task and to introduce nonlinearity to the model, activation functions are added at the end of each layer of the neural network. Figure 1.6 shows examples of the most frequently used activation functions.



Figure

Figure 1.6 – Graphics of activation functions

1.8. Gradient descent

Now that the theory behind the algorithms of computing the result, the forward pass, of a neural network has been discussed, the only thing left is to

understand how they learn. Gradient descent is a widely used optimization algorithm, which is focus on minimizing an error function, propagated through the network. The main idea is to compute partial derivatives with respect to the weights, starting from the end, or the result of the network, and propagating it back through all the layers back to the beginning. This is where the name backpropagation comes from, or as some regard it as backpropagating through time.

In order to update all the weights with gradient descent, it is necessary to have some kind of evaluation of the current output with respect to what was expected. This evaluation is called the cost function. It takes in the result of the forward pass of the network, compares it with what was expected through a differentiable function, and that is used to update all the parameters of the network. By minimizing the cost function we maximize the network accuracy.

What the gradient descent algorithm done, broadly speaking, is it measures the local gradients of the cost function with regard to the parameter vector θ, and it updates them in the direction of descending gradient. The size of the update is determined by the learning rate, which scales the update, making the update norm lower and thus effectively lowering the update to the weights at the given iteration.

Let's analyze the mean-squared error function (MSE) as an example, where we have a single weight vector θ. It is defined the following way:

$$MSE(x, y, \theta) = \frac{1}{n}\sum_{i=0}^{n}(\theta^T \cdot x^{(i)} - y^{(i)})^2$$

Where $\boldsymbol{x}$ is the input, $\boldsymbol{y}$ is the target, and $\boldsymbol{n}$ is the number of instances of $\boldsymbol{x}$ and $\boldsymbol{y}$.

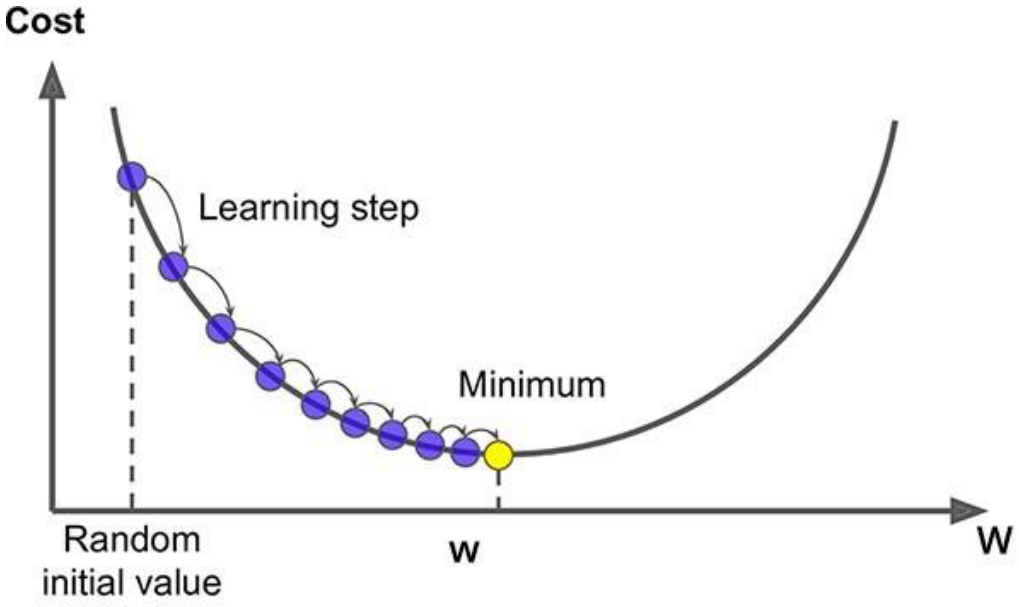


Figure 1.7 – Gradient descent

Figure 1.7 visualizes the basic workings of the gradient descent algorithm. Having calculated the mean squared error, the next step is to update the weights. The next step is to calculate the partial derivatives of the cost function with respect to the weights θ, or using the notation $\frac{\partial}{\partial \theta_i}$, which will indicate how much each weight factors in the cost function, providing the ability to tweak that closer to the expected behavior.

$$\frac{\partial}{\partial \theta_j} MSE(x, y, \theta) = \frac{2}{n} \sum_{i=0}^{n} \left(\theta^T \cdot x^{(i)} - y^{(i)}\right) x_j^{(i)}$$

It can be rewritten in a vectorized form instead of computing each one of the partial derivatives.

$$\nabla_\theta MSE(x, y, \theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(x, y, \theta) \\ \frac{\partial}{\partial \theta_1} MSE(x, y, \theta) \\ \dots \\ \frac{\partial}{\partial \theta_m} MSE(x, y, \theta) \end{pmatrix} = \frac{2}{n} X^T \cdot (X \cdot \theta - y)$$

This process is called batched gradient descent, because the whole input set $\boldsymbol{X}$. One way to regularize the network and reduce the computational complexity of each step is to split the dataset $\boldsymbol{X}$ and their correspondence in $\boldsymbol{Y}$ into chunks of several instances each, which is called stochastic gradient descent. This step will reduce the computations needed to do a single update to the parameters and will ensure, that every step the network is given different sets of samples, making it harder to overfit, thus having a regularization effect. This strategy is called mini-batch gradient descent, and it is employed in every single modern deep neural network. It is virtually impossible to fit the network and the gigabytes, if not terabytes of data in the effective memory of the machine. The only step left is to account for the learning rate $\boldsymbol{\eta}$ and update the parameters. This can be easily done with the following formula.

$$\theta \coloneqq \theta - \eta \nabla_\theta MSE(x, y, \theta)$$

1.9. Adam

Adam, which stands for adaptive moment estimation, is an adaptive optimizer for neural networks, that was proposed by Kingma and Ba. What makes it adaptive is the ability to learn and adapt the learning rate layer-wise. To achieve this, it uses first and second order adaptive gradient estimation to calculate the adaptive learning rates. We define the n-th moment as:

$$m_n = E[X^n]$$

This results in the first moment being the expected value, and the second moment is uncentered variance.

1. $m \leftarrow \beta_1 m - (1 - \beta_1)\nabla_\theta J(\theta)$
2. $s \leftarrow \beta_2 s - (1 - \beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
3. $\hat{m} \leftarrow \frac{m}{1 - \beta_1^T}$
4. $\hat{s} \leftarrow \frac{s}{1 - \beta_2^T}$
5. $\theta \leftarrow \theta + \eta \hat{m} \oslash \sqrt{\hat{s} + \varepsilon}$

Where

$\nabla_\theta J(\theta)$ – gradient of the cost function *J* with respect to weight vector θ.

β_1, β_2 – coefficients set by the user; indicate momentum and scaling decay.

η – learning rate.

ε – extremely small number to avoid division by 0.

These are the 5 steps for calculating and updating the parameters using Adam optimizer. It deviates from the original stochastic gradient descent by bringing in the statistics about the gradients gathered over training. Step 1 and 2 compute the exponentially decaying average of first and second moments of the gradients. Step 3 and 4 make sure that the network is not biased towards 0, because the parameters are set to 0 at the beginning of the training. Step 5 combines the statistics with the gradients to introduce the adaptive learning rate, which is scaled by calculations in steps 3 and 4, making the learning rate higher in the dimension of the steeper slope.

Adam optimizer currently being used in nearly all modern deep neural networks. It has proven to achieve great results with little to no tuning. But if a team has enormous computational capacity, a better convergence can be achieved

by finding the correct parameters for stochastic gradient descent through trial and error.

1.10. Convolutional Neural Network

A convolutional neural network (CNN) is a specialized type of neural networks, which aims at working with grid-like data or with timeseries (audio signals). They emerged from the study on the visual interpretation done by the biological brain, which resulted in achieving state of the art performance on image data. The main mathematical building block of this kind of neural networks is the convolutional layer and the convolution operation. The first layer is not connected to every pixel in the image, but rather on a field of pixels, a receptive field which contextualizes information on blocks, rather than on every single input. The next layer is connected to a field of outputs of the outputs of the first layer. This hierarchical architecture is what leads to the great performance CNNs have on timeseries and images.

CNNs capture the temporal and spatial dependencies in the data. The discrete operation of convolution on vectors $x \in \mathbb{R}^n,\ w \in \mathbb{R}^m$, which change over time *t* is defined as $y(t) = (x * w)(t)$, where $\boldsymbol{x}$ is the inputs, $\boldsymbol{w}$ is the kernel. We can formulate convolution with the following formula:

$$y(t) = (x * w)(t) = \sum_{k=-\infty}^{+\infty} x(k)w(t-k)$$

CNNs are operating on tensors, the reason behind it being that the data size is large and complex computations. Kernel and data is stored separately, so it is necessary to define it this function as 0 everywhere, except the finite set of values we need. With this, we can shorten the range and move to a more practical

implementation of the convolution operation, and it allows us to compute an infinite sum over a finite array. So we can redefine the formulation over a two-dimensional input D and a two-dimensional kernel K as:

$$Y(i,j) = (D * K)(i,j) = \sum_m \sum_n D(m,n)K(i-m, j-n)$$

CNNs are also known to have the shift-invariant property, based on the shared weights architecture of the convolution kernels, sparse interactions, and equivariant representations. As described earlier, convolutional layers interact over a field of values, instead of every single pixel in the case of MLPs, from which the sparsity property arises, by making the kernel smaller than the input image.

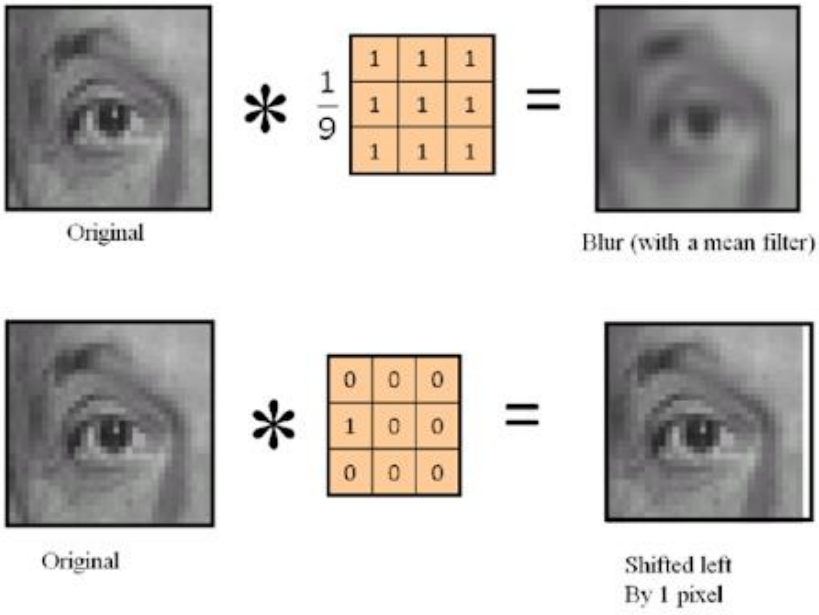


Figure 1.8 - Convolution kernel

Different kernels are used throughout the history of computer visions, with carefully handcrafted kernels transforming the image in certain ways to extract useful information out of the data. In a CNN, the task is to learn a set of kernels, that transforms the image and extracts the features needed in order to maximize a

metric on the given task. A single set of parameters is learned, instead of a different set for every location. This is done by sliding the kernel across the data, and computing the convolution with small kernels. Two additional parameters, that are manually set are padding and stride. Padding of ***k*** simply expands the input matrix by a factor of ***k***, adding ***k*** zero values to each side of the input matrix. Stride ***s*** is the parameter, which defines by how much the kernel slides over the input image. By default, stride is 1, which means that we compute the matrix multiplication of kernel by a patch of the input data, then slide it by ***s*** samples and repeat.

After applying a convolution, the result is a set of linear activations, on top of which a nonlinear activation function is applied, which is done to achieve a nonlinear transformation of the feature space. It has been proven useful and effective to apply a function called the **pooling function.** It is common to use a specific instance of pooling functions called max and average pooling. The idea behind them is simple – return the maximum or mean over a rectangular neighborhood.

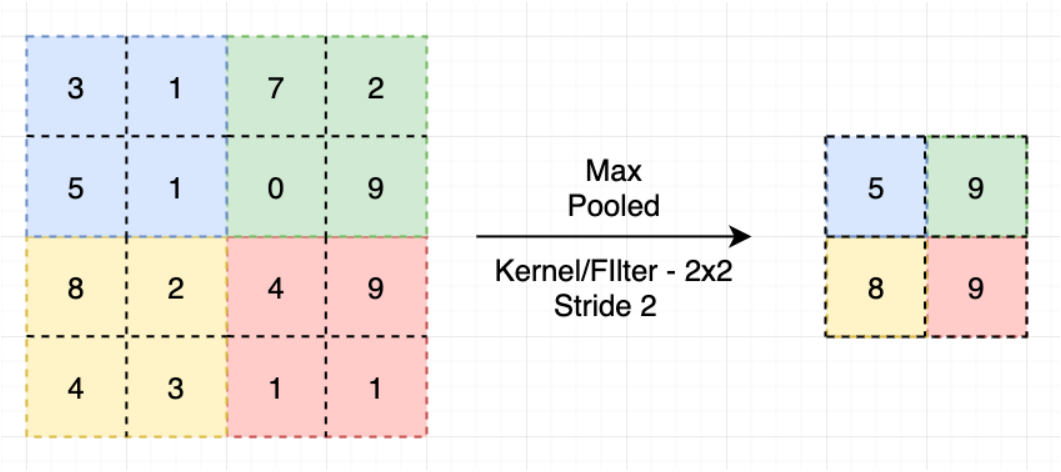


Figure 1.9 – pooling layer

The goal of a pooling layer ss to subsample the input data and reduce the computational load, memory usage, and number of parameters. It also introduces

some level of invariance to small changes within the outputs of the convolutional layers. By inserting a pooling layer after each layer, it is possible to achieve a translation invariance on a larger scale. As an example, if we shift the image by one pixel to some direction, we will still receive roughly the same outputs exactly because of the pooling layers giving the ability to remove this dependency. Although this operation should be done with the task in mind. This is a destructive action, that removes 75% of the outputs of each convolution layer. In a task such as semantic segmentation, where pixel-perfect masks are necessary, it is impractical to use pooling layers as the information about the whole image is needed to make a prediction, which would result in an over smoothing of the predicted mask with when applying pooling. But on the other hand, if a task of image classification is taken, there are little to no downsides to pooling, as it reduces the time and memory complexity of the network. All these concepts come together to create a convolutional neural network.

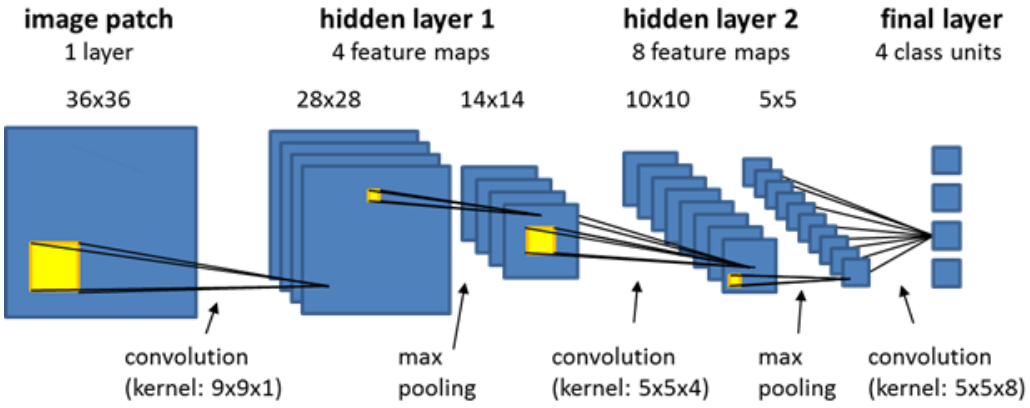


Figure 1.10 – convolutional neural network

1.11. Transposed convolution layers

Convolutions have been discussed as the method to lower the dimension of the input and extract features, but what if expanding the input is the goal, instead of contracting it? This is where transposed convolutions come in. A simple use-case would be in the task of super-resolution. The task, where a small input is given, and a larger, higher quality output is expected, in other words the task of upsampling. Which is already similar to the task at hand.

The transposed convolution layer is equivalent to stretching the image by inserting empty rows and columns with zero values, then performs regular convolution on top of that. In this layer, the stride parameter defines how much the input will be stretched. The larger the stride, the larger the output.

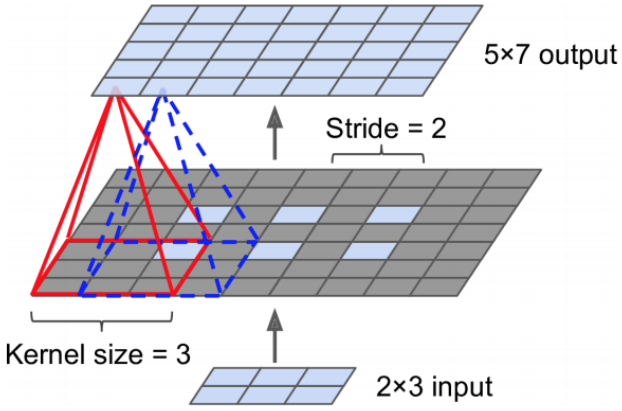


Figure 1.11 – transposed convolution

1.12. Batch Normalization

Normalization is a technique in machine learning, which aim at making different sample be closer to each other in the feature space. This is done to help the model generalize, as different samples are closer to each other making it harder to overfit. While training, every mini-batch might have a slightly different

distribution in-between layers, thus making it harder for the network to learn, making the network chase the moving target. This change in distribution between layers is called internal covariate shift.

Batch Normalization is a type of normalization, used in neural networks. It has been introduced in a 2015 paper, authored by Sergey Ioffe and Christian Szegedy. This kind of normalization adaptively normalizes the data in batches to look more like other different samples, which leads to faster convergence and often lower number of parameters necessary for the task. The main advantage is the stimulation of gradient flow through deep neural networks and battles the problem of exploding/vanishing gradients. The algorithm is simple, it zero-centers and standardizes the data, then it scales it and shift by the $\boldsymbol{y}$ and $\boldsymbol{\beta}$ learnable parameters. Meaning that the model has the capacity to learn them for a better representation, but as the parameters are learnable, the model can even learn to ignore the normalization overall. The estimation of the mean and standard deviation is done over the whole batch, where the name comes from. The full algorithm is as follows.

$$\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} x^{(i)}$$

$$\sigma_B^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(x^{(i)} - \mu_B\right)^2$$

$$\widehat{x^{(i)}} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

$$z^{(i)} = \gamma \otimes \widehat{x_i} + \beta$$

Where:

μ_B vector of means, evaluated over a mini-batch B.

σ_B^2 vector of variances, evaluated over a mini-batch B.

m_B number of samples in the mini-batch B.

$\widehat{x^{(i)}}$ zero-centered and normalized input vector for sample i.

γ learnable scale parameter for the layer.

β learnable shift parameter for the layer.

$z^{(i)}$ output of Batch Normalization for instance i.

After a model is trained, there comes an issue. What should be done after it is moved on to inference? The first action that can be taken is the fusion of layers with batch normalization. As it's a simple operation, it is easily possible to combine convolution layers with batch normalization, thus reducing the amount of parameters and complexity. Then, comes the question, what if during test time a single instance is received, instead of a batch? Most implementations of Batch Normalization track a moving average of the input means and variances during training, which is then used during inference as the estimated layer statistics. Ioffe and Szegedy have demonstrated in their paper, that Batch Normalization has a significant improvement in all deep neural networks.

1.13. Skip connections

Kaiming He. et al. won the ILSVRC 2015 challenge using a novel technique in their Residual Network, achieving a 3.5% in top 5 accuracy lead over their competition. They were able to achieve this by creating an extremely deep CNN composed of 152 layers. In normal conditions, training such a deep neural network will give little to no results due to vanishing gradients when training too long, as the signal will not be strong enough to be propagated back such a long way. This is

where their invention of skip connections comes in. Another key change is giving the ability to let the signal pass through the network without going through the nonlinearity of activation functions. The nonlinear function, by their nature, are the cause of exploding/vanishing gradients.

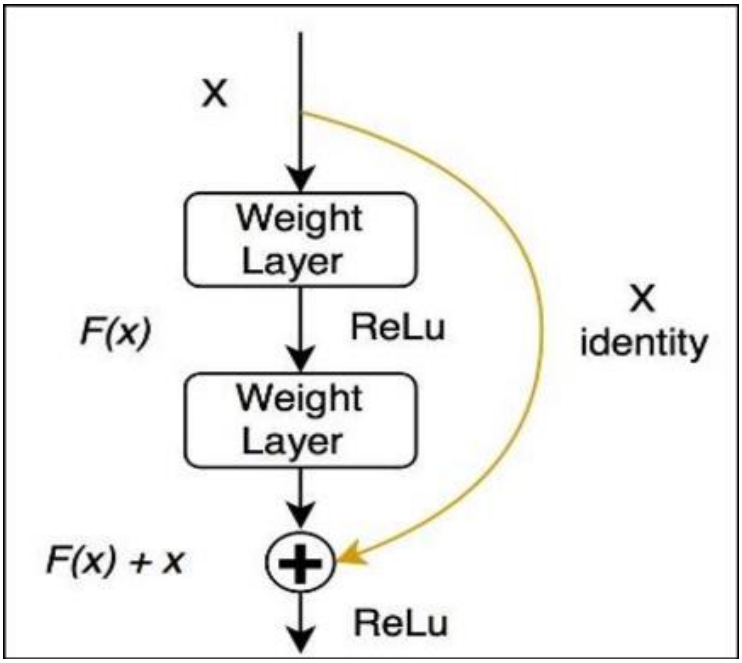


Figure 1.12 – skip connection

The idea is simple and is demonstrated in Figure 1.12. While passing through the network, a copy of the input is created, multiplied by the identity matrix and saved. Then the other copy is being fed to other layers. It is important that the result of these layers has the same dimensions, as the previous copy. Afterwards, the two results are added together and passed through an activation function. It is common to add a Batch Normalization layer after several layers, but before adding both of them in. It was done by the authors and is illustrated in Figure 1.13. By doing this, it is possible to further refine and normalize the network. This simple trick speeds up training significantly and leads to a better and faster convergence, even in extremely deep neural networks.

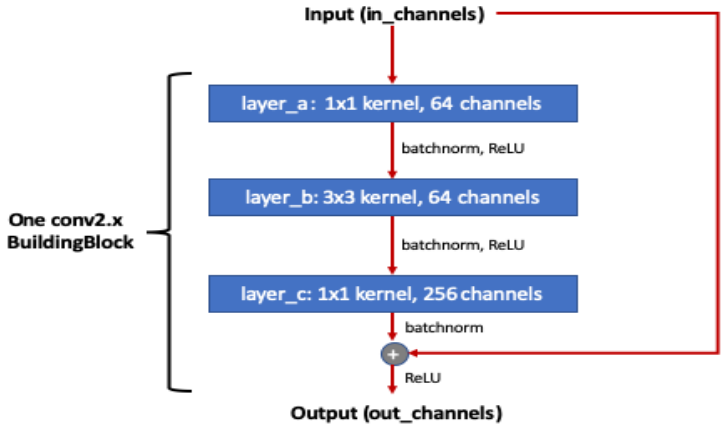


Figure 1.13 – ResNet block

1.14 Multi-resolution STFT loss

Loss function lie at the heart of optimization. They are what determines how good the model is doing in terms of solving the task, which is inevitably minimized. Given that the task lies with dealing with a continuous digital audio signal, where a single second is characterized by thousands of samples, it is not trivial to create a suiting loss function. A simple absolute difference between the generated and target audio signal will not capture long temporal dependencies and correlation in small patches of audio. This is why something that takes in temporal features into consideration is necessary, to create a system that not just generates something similar, but is actually perceived as a similar sounding audio.

Yamamoto, Song, and Kim proposed a new approach to training with audio sequences. They introduced a multi-resolution STFT loss, at the basis of which lies the STFT transformation. A single STFT loss can be defined as:

$$L_s(G) = E_{z\sim p(z), x\sim p_{data}}[L_{sc}(x, \hat{x}) + L_{mag}(x, \hat{x})]$$

Where x is the target audio signal, $\hat{x}$ is the generated audio signal by neural network *G*, L_{sc} and L_{mag} denote spectral convergence and magnitude respectively. The latter two are defined as:

$$L_{sc}(x, \hat{x}) = \frac{\| \, | \, STFT(x) \, | - | \, STFT(\hat{x}) \, | \, \|_F}{| \, STFT(x) \, |_F}$$

$$L_{mag}(x, \hat{x}) = \frac{1}{N} \| \log|STFT(x)| - \log|STFT(\hat{x})| \, \|_1$$

In spectral convergence, Frobenius norm is used, and in log magnitude loss, L1 norm is being used, which is the sum of absolute values. Now the only thing that is left is to assemble the STFT losses with different parameters such as hop size, number of fft bins and window length into a single loss. This is what comes out.

$$L(G) = \frac{1}{M} \sum_{m=1}^{M} L_s^{(m)}(G)$$

The reason why multiple STFT losses are necessary is because the network can overfit to a single set of STFT parameters, or a single representation. This is bad, because the change in their parameters leads to slightly different representations; e.g., increasing window size gives higher frequency resolution while reducing temporal resolution. This is why it is important to have several STFT losses to create a model that is able to create realistic and natural sounding audio.

Section 2. Experiments and analysis

2.1. Experiment requirements

- Python 3 programming language
- PyTorch as the library of choice for Neural Networks and training.
- Seaborn, matplotlib, and IPython for visualization
- ResamPy as the library for resampling

2.2. Data preprocessing

The task at hand is to upsample an audio signal from 22050 sample rate up to 44100, effectively doubling it. All that is needed for this is a dataset of 44100 audio. Audio is normalized by volume and projected onto [-1,1] scale. The dataset that is being used is VCTK multi-speaker dataset, which contains 48kHz audio. It is downsampled to 44100 and used as a target. While assembling the batch, with the help of resampy it is easy to downsample a high quality 44100 rate signal to a lower quality, lower rate 22050 audio signal. It is necessary to constrain the audio signal time-wise, as it will not fit into memory if we use several minutes of audio for each entry in the batch. 2 seconds samples are chosen as the size of each entry in the batch. So the audio is loaded and a random chunk of 2 seconds (44100 samples) from the downsampled 22050 signal, and that the target is 88200 samples. A pair of downsampled and its corresponding upsampled segments are formed. After receiving the pair, the 22050 rate part is upsampled using the fast Kaiser filter to 44100 rate. It will be discussed further on.

2.3. Baseline model – Kaiser upsampling

In digital signal processing, one of the best ways to upsample a signal is using a Kaiser filter. It's an interpolation method, that in our case inserts values in between the each two samples. Then it estimates the values of the inserted values to best interpolate them for a good sounding audio. Kaiser filters are generated using the windowed method of finite impulse response (FIR) filter design with the use of a Kaiser window. A Kaiser window is generated through the use of Bessel functions with the length and shape parameters. By varying these parameters the main and side lobes can be adjusted accordingly. It maximizes the energy concentration on the main lobe, but this is often expensive to compute. The Kaiser window is defined as:

$$\omega_0(x) \triangleq \begin{cases} \dfrac{I_0[\pi\alpha\sqrt{1-\left(\frac{2x}{L}\right)^2}]}{I_0[\pi\alpha]}, & |x| \leq \dfrac{L}{2} \\ 0, & |x| > \dfrac{L}{2} \end{cases}$$

Where I_0 is the 0^{th} order modified Bessel function, L is the window duration, α is the nonnegative value that determines the shape of the window.

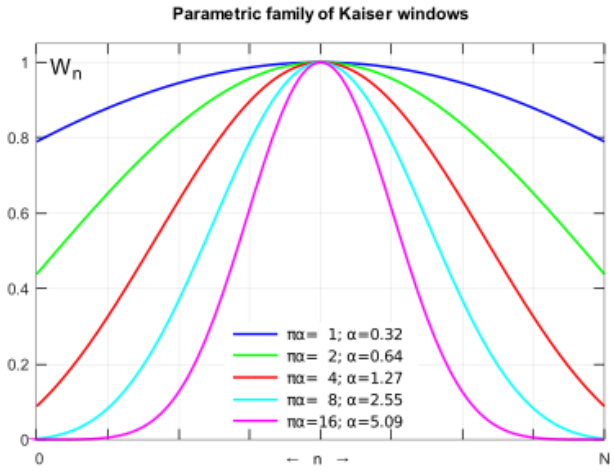


Figure 2.1 – Kaiser windows constructed with different parameters

2.4. Building U-Net Neural Network

The network that will be used is a U-Net, which can be used to work with the task of super-resolution. The main idea behind this network is to take in some poor approximation of the desired signal and enhance that, making it a better approximation closer to the intended, higher quality and higher sample rate audio.

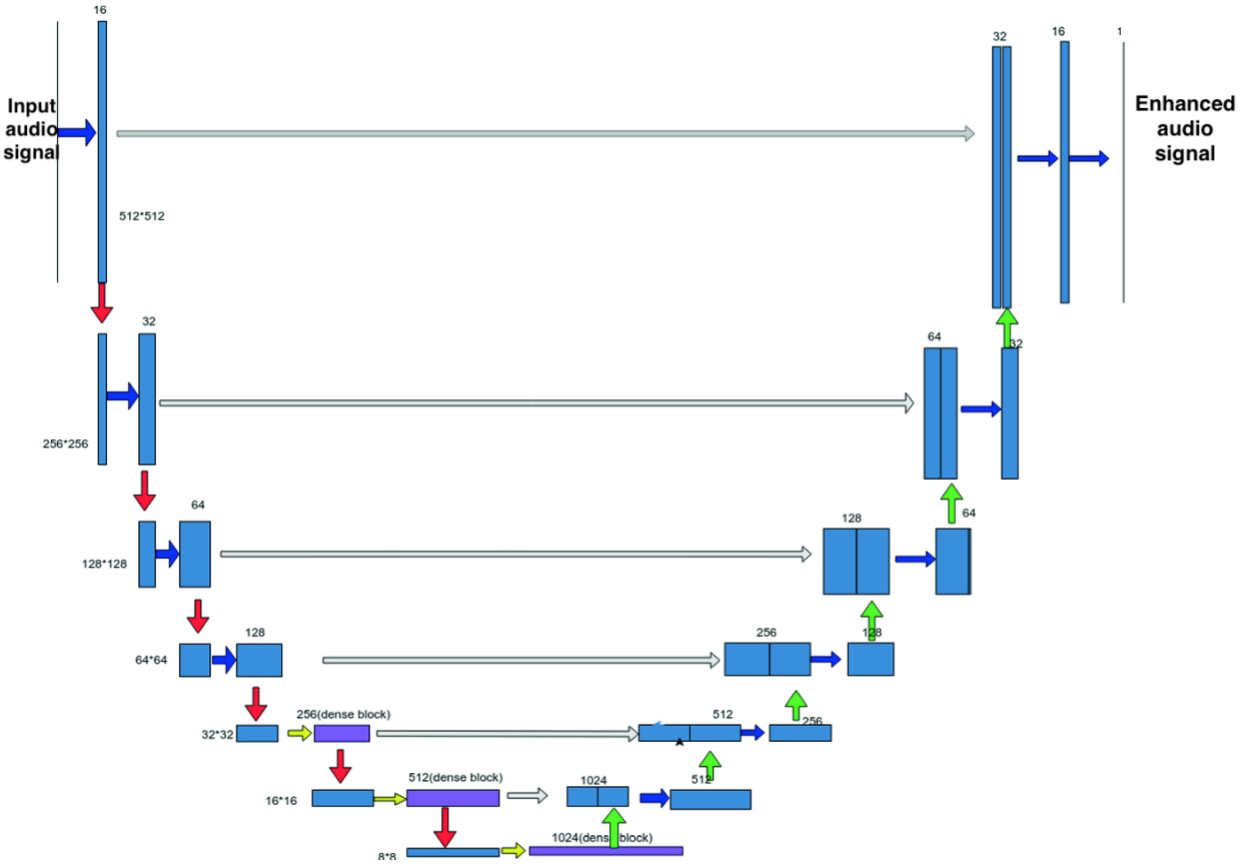


Figure 2.2 – U-Net neural network

The architecture in Figure 2.2. The blue arrows are a double convolution block. They consist of 1D convolution with kernel size 3 and padding 1, followed by a Batch Normalization layer and a ReLU activation function. Then it is fed to the second 1D convolution layer with kernel size 3 and padding 1. The red arrow

displays, that a stride of 2 is being used in the previous double-convolution block. The gray arrows are the residual connections. The green arrows display an upsampling block, which takes in the lower dimension representation, linearly upsamples it twice the size it was, concatenates it with the residual connection that is coming in and that is followed by a blue arrow, which is the double-convolution block but with stride set to 1, while we are upsampling, or going up. The task of creating a feature map with the help of CNNs is simple, but it is extremely hard for them to reconstruct an image back from that feature map.
The whole idea of a U-Net revolves around this problem and around solving it. Providing the copy of all the outputs of the intermediate steps while upsampling is of great help and improves the quality and convergence significantly.

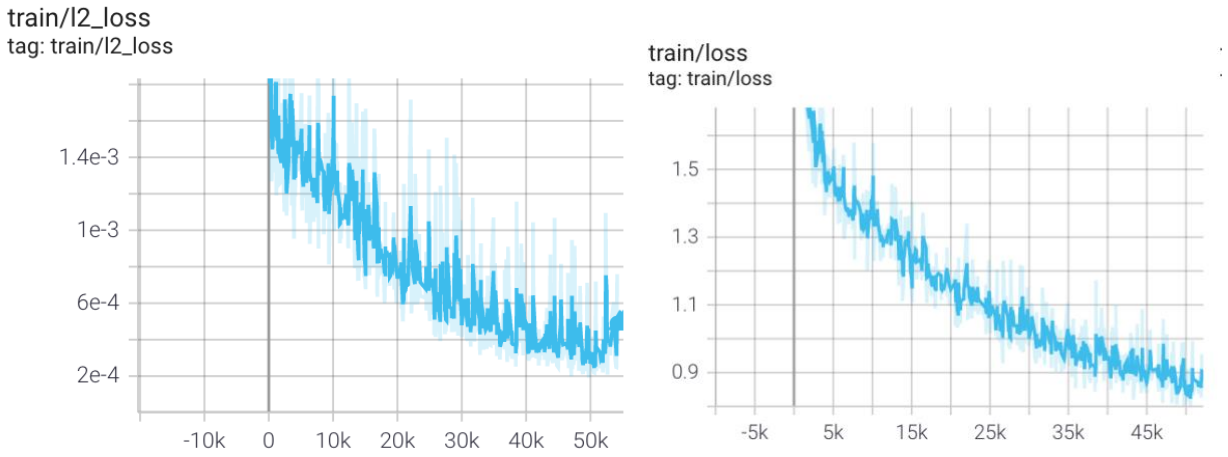
2.5. Training objective and optimization

Now that the data is processed and the model is built, the next step is to define the training objective and optimization. The U-Net acts as an enhancer network. The fastest upsampling using Kaiser fast filter is used to produce a rough estimate of the upsampled audio signal, which has a low quality, but is computationally inexpensive working much faster than real-time. Having the rough and poor quality estimate of the upsampled audio, the network is now tasked with enhancing it, adding in more details in the higher and lower frequencies, making it perceptually better. Given the pairs of poor quality upsampled and original audio in high sample rate, pairs are formed on which the model is trained. Multi-resolution STFT loss will be used in pair with MSE loss. 3 resolution of the multi-resolution STFT loss will be used with the following parameters:

STFT loss	FFT bins	Window size	Hop size
$L_s^{(1)}$	512	240	50
$L_s^{(2)}$	1024	600	120
$L_s^{(3)}$	2048	1200	240

Table 1 – Multi-resolution STFT parameters

Given that the signal is in the range [-1,1], the MSE loss will be significantly lower than multi-resolution STFT (MR-STFT) loss. This is why it is scaled by a coefficient of 100, to provide an importance to the training procedure, but the MR-STFT loss is still more important as it is partly a perceptual loss, unlike MSE. The optimization is done with Adam, with the learning rate set to 0.0001, weight decay set to 0.00001, betas are the default at $\beta_1 = 0.9, \beta_2 = 0.999$. At each training step, the gradient norm is clipped at 10 to avoid exploding gradients and stabilize the training.



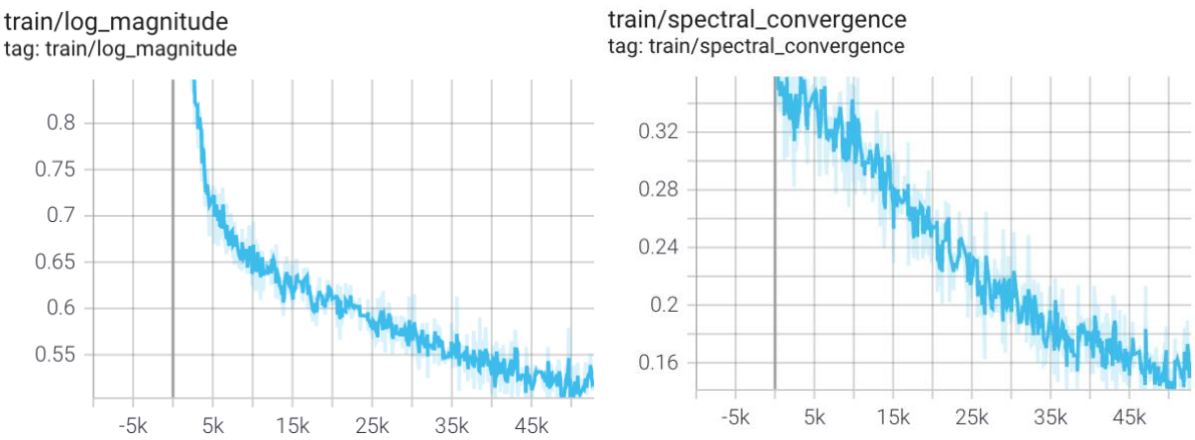


Figure 2.3 – training losses a) MSE loss; b) total training loss as the sum of all the losses; c) log magnitude loss of the MR-STFT; d) spectral convergence loss of the MR-STFT loss.

As shown on Figure 2.3, the training converged quickly and although it might look like it could have trained more, the validation loss, started to grow on one of the most important losses – the spectral convergence loss. It can be seen that all the losses started to plateau and even slightly grow, which is the second reason to stop, before overfitting or starting to diverge.

2.6. Validation

Validation is done on audios that the model has never seen, making it a fair comparison and an evaluation of generalization. All the same losses are calculated during validation, training and gradient flow is turned off, weight norms are removed. This is all to provide a fair comparison to never-seen examples.

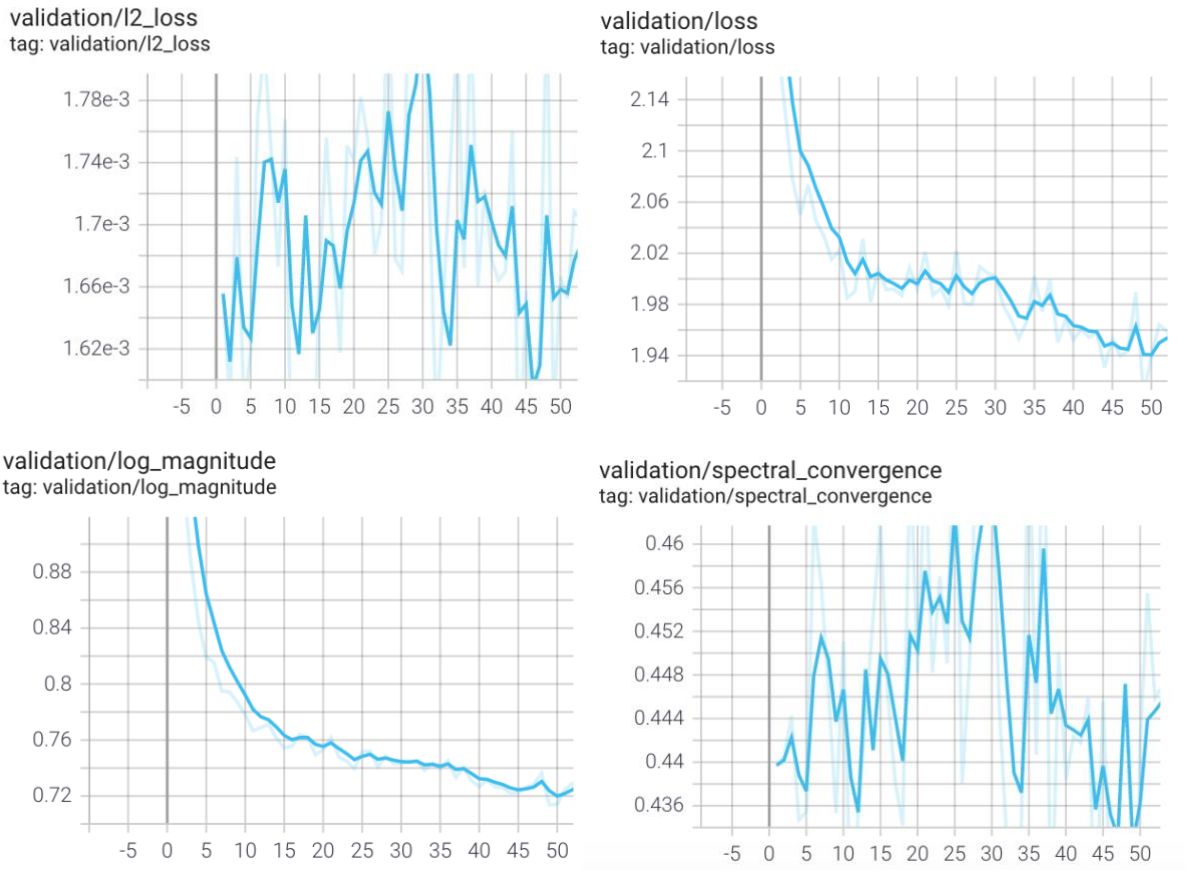


Figure 2.4 – validation losses a) MSE loss; b) total training loss as the sum of all the losses; c) log magnitude loss of the MR-STFT; d) spectral convergence loss of the MR-STFT loss.

In Figure 2.4 we can see that all the losses are healthy, and towards the end spectral convergence starts to rise, which is the reason to stop training. It is one of the most important losses, as it is responsible for measuring the quality of generation of the higher frequencies, which is the hardest part of generating audio.

2.7. Analyzing and comparing results

Now that there is a trained network, it is time to analyze the results of it and compare it to the best results achievable with methods from digital signal processing, which do not involve Neural Networks. The baseline is the Kaiser filter, which has a fast and the best quality implementations. The difference between them are that uses more coefficients to compute the window and estimate the missing samples in a signal.

In order to compare all these methods, it is necessary to create an evaluation function. Two classical approaches from digital signal processing, are signal-to-noise ratio and log spectral distance. These are the functions we will be evaluating with, as well as time complexity.

$$SNR(x, y) = 10 \log \frac{\|y\|_2^2}{\|x - y\|_2^2}$$

Where ***x*** and ***y*** are the generated and target audio signals.

$$LSD(x, y) = \frac{1}{L} \sum_{l=1}^{L} \sqrt{\frac{1}{K} \sum_{k=1}^{K} \big(Y(l, k) - X(l, k)\big)^2}$$

$$X = \log|STFT(x)|^2 , Y = \log|STFT(y)|^2$$

Where ***L*** and ***K*** are the number of time frames and frequency bins respectively. ***X*** and ***Y*** are the log power spectrogram of their respective signals. 2048 fft bins and hop size of 512 is being used while calculating STFT.

Objective	Kaiser fast upsampling	Kaiser best upsampling	U-Net upsampling
SNR ↑	6.091	6.165	11.067
LSD ↓	6.272	6.119	2.6916

Table 2 – Evaluation results. U-Net against classical methods.

As indicated by the arrows, a higher SNR is desirable and a lower log spectral distance. The method with U-Net outperforms classical methods by a large margin on both the evaluation functions. All the evaluation has been done on the test set, which was not seen by the model during training, which gives a good reason to think that the model generalizes well to unseen audio, greatly outperforming best classical methods.

Objective	Kaiser fast upsampling	Kaiser best upsampling	U-Net upsampling
Upsampling time ↑ (Real-time ratio)	170	22.8	55.5

Table 3 – upsampling time comparison between methods

Table 3 display the generation time, calculated as the real-time ratio. This means the amount of seconds that can be upsampled, in a second of computation, so the higher the number the better. Although Kaiser fast is the fastest algorithm, the quality on it is the worst of the three. And Kaiser best, which is focused on quality, takes twice as long to upsample an audio, in comparison to the U-Net.

Conclusion

Machine learning and deep learning is on the rise, bringing improvements and the states of the art in multitudes of fields. In this work, the focus lies on the task of audio super-resolution, which is genearating a higher quality higher sample rate audio out of a poor quality audio signal. Different current best and fast implementations of upsampling methods are presented. A neural network is built, trained, and presented, which greatly outperforms the previous best upsampling methods, that rely on windows for upsampling. Overall, the model is successful, because it exceeds in quality greatly and computational requirements in comparison to the best classical methods. Even having such great success, there is certainly room to grow. The loss functions showed, that that the network could be further improved by bringing in other training techniques such as weight regularization, correct initialization and making the network deeper. Another interesting and promising way of future development would be to try train it as a Generative Adversarial Network or completely overhaul the architecture and explore lighter architecture, than the one presented.

Literature

[1] Aurelien Geron. [book] Hand-On Machine Learning with Scikit-Learn and TensorFlow.

[2] Ian Goodfellow, Yoshua Bengio, Aaron Courvile. [book]Deep Learning.

[3] Sebastian Raschka. [book] Python Machine Learing.

[4] Professor Dr. W. Brauer [article] Untersuchungen zu dynamischen neuronalen Netzen. http://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf

[5] [website] THE SHORT-TIME FOURIER TRANSFORM. https://www.dsprelated.com/freebooks/sasp/Short_Time_Fourier_Transform.html

[6] Ricardo Gutierrez-Osuna[article] Short-time Fourier analysis and synthesis. http://research.cs.tamu.edu/prism/lectures/sp/l6.pdf

[7] Kartik Chaudhary [article] Understanding Audio data, Fourier Transform, FFT and Spectrogram features for a Speech Recognition System. https://towardsdatascience.com/understanding-audio-data-fourier-transform-fft-spectrogram-and-speech-recognition-a4072d228520

[7] Ryuichi Yamamoto, Eunwoo Song and Jae-Min Kim [article] PARALLEL WAVEGAN: A FAST WAVEFORM GENERATION MODEL BASED ON GENERATIVE ADVERSARIAL NETWORKS WITH MULTI-RESOLUTION SPECTROGRAM. https://arxiv.org/pdf/1910.11480.pdf

[8] Nigel Redmon [article] A closer look at upsampling filters. https://www.earlevel.com/main/2010/12/11/a-closer-look-at-upsampling-filters/

[9] Kajal Mishra [article] Audio Signal Processing- Understanding Digital & Analog Audio Signal Processing. https://www.pathpartnertech.com/audio-signal-processing-understanding-digital-analog-audio-signal-processing/

[10] Ronneberger et al.[article] U-Net: Convolutional Networks for Biomedical

Image Segmentation. https://arxiv.org/pdf/1505.04597.pdf

[11] Diederik P. Kingma, Jimmy Lei Ba [article] ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. https://arxiv.org/pdf/1412.6980.pdf

[12] Krizhevsky et al. [article] ImageNet Classification with Deep Convolutional Neural Networks. https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

Appendix A. Program Text

```
import torch
from torch import nn
import torch.nn.functional as F
import numpy as np
import resampy
from torch.utils.data import Dataset
import random
import soundfile as sf
from tqdm import tqdm
import argparse
import json
import os
import sys
from torch.optim import Adam
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.double_conv = nn.Sequential(
            nn.Conv1d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm1d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv1d(
                out_channels, out_channels, kernel_size=3, padding=1, stride=stride
            ),
            nn.BatchNorm1d(out_channels),
```

```
        )
    def forward(self, x):
        return self.double_conv(x)

class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv1d(in_channels, out_channels, kernel_size=1)
    def forward(self, x):
        return self.conv(x)
class UpscaleBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.up = nn.Upsample(scale_factor=2, mode="linear", align_corners=True)
        self.conv = ConvBlock(in_channels, out_channels)
    def forward(self, x1, x2):
        x1 = self.up(x1)
        diffY = x2.size()[1] - x1.size()[1]
        diffX = x2.size()[2] - x1.size()[2]
        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2, diffY // 2, diffY - diffY // 2])
        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)
class UNet(nn.Module):
    def __init__(self):
        super(UNet, self).__init__()
        self.inp = ConvBlock(1, 16)
        self.down1 = ConvBlock(16, 32, stride=2)
        self.down2 = ConvBlock(32, 64, stride=2)
        self.down3 = ConvBlock(64, 128, stride=2)
        self.down4 = ConvBlock(128, 256, stride=2)
        self.down5 = ConvBlock(256, 512, stride=2)
        self.down6 = ConvBlock(512, 1024, stride=2)
        self.down7 = ConvBlock(1024, 2048 // 2, stride=2)
        self.up1 = upscale_block(2048, 1024 // 2)
        self.up2 = upscale_block(1024, 512 // 2)
        self.up3 = upscale_block(512, 256 // 2)
        self.up4 = upscale_block(256, 128 // 2)
```

```
        self.up5 = upscale_block(128, 64 // 2)
        self.up6 = upscale_block(64, 32 // 2)
        self.up7 = upscale_block(32, 16)
        self.final = OutConv(16, 1)
    def forward(self, signal):
        x1 = self.inp(signal)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x6 = self.down5(x5)
        x7 = self.down6(x6)
        x8 = self.down7(x7)
        x = self.up1(x8, x7)
        x = self.up2(x, x6)
        x = self.up3(x, x5)
        x = self.up4(x, x4)
        x = self.up5(x, x3)
        x = self.up6(x, x2)
        x = self.up7(x, x1)
        res= self.final(x)
        return signal + res

def stft(x, fft_size, hop_size, win_length, window):
    x_stft = torch.stft(x, fft_size, hop_size, win_length, window)
    real = x_stft[..., 0]
    imag = x_stft[..., 1]
    return torch.sqrt(torch.clamp(real ** 2 + imag ** 2, min=1e-7)).transpose(2, 1)

class SpectralConvergengeLoss(torch.nn.Module):
    def __init__(self):
        super(SpectralConvergengeLoss, self).__init__()
    def forward(self, x_mag, y_mag):
        return torch.norm(y_mag - x_mag, p="fro") / torch.norm(y_mag, p="fro")

class LogSTFTMagnitudeLoss(torch.nn.Module):
    def __init__(self):
```

```
        super(LogSTFTMagnitudeLoss, self).__init__()
    def forward(self, x_mag, y_mag):
        return F.l1_loss(torch.log(y_mag), torch.log(x_mag))
class STFTLoss(torch.nn.Module):
    """STFT loss module."""

    def __init__(self, fft_size=1024, shift_size=120, win_length=600, window="hann_window"):
        super(STFTLoss, self).__init__()
        self.fft_size = fft_size
        self.shift_size = shift_size
        self.win_length = win_length
        self.window = getattr(torch, window)(win_length)
        self.spectral_convergenge_loss = SpectralConvergengeLoss()
        self.log_stft_magnitude_loss = LogSTFTMagnitudeLoss()

    def forward(self, x, y):
        x_mag = stft(x, self.fft_size, self.shift_size, self.win_length, self.window)
        y_mag = stft(y, self.fft_size, self.shift_size, self.win_length, self.window)
        sc_loss = self.spectral_convergenge_loss(x_mag, y_mag)
        mag_loss = self.log_stft_magnitude_loss(x_mag, y_mag)

        return sc_loss, mag_loss


class MultiResolutionSTFTLoss(torch.nn.Module):

    def __init__(self,
                 fft_sizes=[1024, 2048, 512],
                 hop_sizes=[120, 240, 50],
                 win_lengths=[600, 1200, 240],
                 window="hann_window"):
        super(MultiResolutionSTFTLoss, self).__init__()
        assert len(fft_sizes) == len(hop_sizes) == len(win_lengths)
        self.stft_losses = torch.nn.ModuleList()
        for fs, ss, wl in zip(fft_sizes, hop_sizes, win_lengths):
            self.stft_losses += [STFTLoss(fs, ss, wl, window)]
```

```
    def forward(self, x, y):
        sc_loss = 0.0
        mag_loss = 0.0
        for f in self.stft_losses:
            sc_l, mag_l = f(x, y)
            sc_loss += sc_l
            mag_loss += mag_l
        sc_loss /= len(self.stft_losses)
        mag_loss /= len(self.stft_losses)

        return sc_loss, mag_loss
class Collater(object):
    def __call__(self, batch):
        y_batch, x_batch = [], []
        for idx in range(len(batch)):
            x, y = batch[idx]
            x_batch += [x.astype(np.float32).reshape(-1, 1)]
            y_batch += [y.astype(np.float32).reshape(-1, 1)]
        x_batch = torch.FloatTensor(np.array(x_batch)).transpose(2, 1)
        y_batch = torch.FloatTensor(np.array(y_batch)).transpose(2, 1)
        return x_batch, y_batch
class AudioDataset(Dataset):
    def __init__(
        self,
        prompts,
        audio_length_threshold=50000,
        source_sample_rate=22050,
        target_sample_rate=44100,
        output_num_seconds=2,
    ):
        self.source_sample_rate = source_sample_rate
        self.target_sample_rate = target_sample_rate
        self.output_num_seconds = int(output_num_seconds)
        audio_files = self.read_prompts(prompts)
        audio_files, audio_lengths = self.get_audio_lengths_and_filter_by_sr(audio_files)
        self.audio_files, self.audio_lengths = self.filter_out_audio_by_length(
            audio_files, audio_lengths, audio_length_threshold
```

```
    )
def read_prompts(self, prompts):
    audio_files = []
    with open(prompts, "r") as prompts:
        data = prompts.readlines()
    for point in data:
        point = point.strip()
        audio_files.append(point.split("|"))
    return audio_files
def get_audio_lengths_and_filter_by_sr(self, audio_files):
    audio_lengths = []
    audio_files = []
    print("Reading audio files and filtering by sr")
    for audio_paths in tqdm(audio_files):
        data_44, sr_44 = sf.read(audio_paths[0])
        data_44_len = len(data_44)
        data_22, sr_22 = sf.read(audio_paths[1])
        data_22_len = len(data_44)
        audio_files.append(audio_paths)
        audio_lengths.append(len(data_44))
    return audio_files, audio_lengths
def filter_out_audio_by_length(self, audio_files, audio_lengths, min_audio_length):
    idxs = range(len(audio_files))
    filtered_audio_files = []
    filtered_audio_length = []
    for idx in idxs:
        filtered_audio_files.append(audio_files[idx])
        filtered_audio_length.append(audio_lengths[idx])
    return filtered_audio_files, filtered_audio_length
def __getitem__(self, idx):
    start_ind = random.randint(
        0,
        self.audio_lengths[idx]
        - (self.output_num_seconds + 1) * self.target_sample_rate,
    )
    source_num_samples = self.output_num_seconds * self.source_sample_rate
    target_num_samples = self.output_num_seconds * self.target_sample_rate
```

```
        if start_ind % 2 != 0:
            start_ind += 1
        target_audio = sf.read(
            self.audio_files[idx][0], frames=target_num_samples, start=start_ind
        )[0]
        source_audio, sr = sf.read(self.audio_files[idx][1], frames=source_num_samples, start=start_ind // 2)
        source_audio = resampy.resample(
            source_audio, sr, self.target_sample_rate, filter="kaiser_fast"
        )
        return (source_audio, target_audio)

    def __len__(self):
        return len(self.audio_files)

def save_checkpoint(model, model_optimizer, iteration, epoch, output_dir):
    filepath = output_dir+'/checkpoint_'+str(iteration)+'.pt'
    print("Saving model and optimizer state at iteration {} to {}".format(
          iteration, filepath))
    torch.save({'model': model.state_dict(),
                'optimizer': model_optimizer.state_dict(),
                'iteration': iteration,
                'epoch':epoch}, filepath)

def load_checkpoint(model, optimizer, checkpoint_path):
    checkpoint = torch.load(checkpoint_path, map_location='cpu')
    iteration = checkpoint['iteration']
    model.load_state_dict(checkpoint['model'])
    optimizer.load_state_dict(checkpoint['optimizer'])
    epoch = checkpoint['epoch']
    return model, optimizer, iteration, epoch


def validate(model, mseloss, stft_loss, writer, epoch, iteration):
    collater = Collater()
    val_set = AudioDataset(
            data_config['val_prompts']
        )
```

```
    val_loader = DataLoader(val_set, num_workers=data_config["num_workers"],
                            shuffle=False,
                            collate_fn=collater,
                            batch_size=train_config["batch_size"] // 2)
    model.eval()
    gen_mse  = 0.0
    gen_mag  = 0.0
    gen_sc   = 0.0
    gen_loss = 0.0
    with torch.no_grad():
        for j in tqdm(range(30)):
            for i, batch in enumerate(val_loader):
                x, y = batch
                x = x.cuda()
                y = y.cuda()
                y_gen = model(x)
                mse = mseloss(y_gen, y)
                sc_loss, mag_loss = stft_loss(y_gen.squeeze(1), y.squeeze(1))
                generator_loss = mse + sc_loss + mag_loss
                gen_mse += mse.data
                gen_sc += sc_loss.data
                gen_mag += mag_loss.data
                gen_loss += rgenerator_loss.data
    steps = (j + 1) * (i + 1)
    gen_mse  = gen_mse  / steps
    gen_sc   = gen_sc   / steps
    gen_mag  = gen_mag  / steps
    gen_loss = gen_loss / steps
    writer.add_scalar('validation/mse_loss', gen_mse, global_step=epoch)
    writer.add_scalar('validation/spectral_convergence', gen_sc, global_step=epoch)
    writer.add_scalar('validation/log_magnitude', gen_mag, global_step=epoch)
    writer.add_scalar('validation/loss', gen_loss, global_step=epoch)
    model.train()
    torch.cuda.empty_cache()
    print(f"Validation {epoch} finished!")

def train(checkpoint_path):
```

```
    train_set = AudioDataset(
          data_config['train_prompts']
        )
    collater = Collater()
    train_loader = DataLoader(train_set, num_workers=data_config["num_workers"],
batch_size=train_config['batch_size'],  collate_fn=collater)
    model = UNet()
    model = model.cuda()
    mseloss = torch.nn.MSELoss()
    stft_loss = MultiResolutionSTFTLoss()
    optimizer = Adam(model.parameters(), lr=model_config["lr"], weight_decay=model_config["weight_decay"])
    checkpoint_dir = train_config["checkpoint_dir"]
    from torch.utils.tensorboard import SummaryWriter
    logs_dir = os.path.join(checkpoint_dir, 'logs')
    if not os.path.exists(logs_dir):
       os.makedirs(log_dir)
    if not os.path.isdir(checkpoint_dir):
       os.makedirs(checkpoint_dir)
    writer = SummaryWriter(logs_dir)
    if checkpoint_path != '':
       model, optimizer, iteration, epochs = load_checkpoint(model, optimizer, checkpoint_path)
    else:
       iteration = 0
       epochs = 1
    for epoch in range(epochs, train_config["epochs"]):
       for repeat in tqdm(range(train_config["data_repeat_times"])):
          exit = False
          for batch in train_loader:
             optimizer.zero_grad()
             x, y = batch
             x = x.cuda()
             y = y.cuda()
             y_gen = model(x)
             gen_mse = mseloss(y_gen, y)
             sc_loss, mag_loss = stft_loss(y_gen.squeeze(1), y.squeeze(1))
             generator_loss = model_config["mse_weight"] * gen_mse + sc_loss + mag_loss
```

```
            generator_loss.backward()
            optimizer.step()
            reduced_gen_sc = sc_loss.data
            reduced_gen_mag = mag_loss.data
            reduced_gen_mse = gen_mse.data
            reduced_gen_loss = generator_loss.data
            writer.add_scalar('train/mse_loss', reduced_gen_mse, global_step=iteration)
            writer.add_scalar('train/spectral_convergence', reduced_gen_sc, global_step=iteration)
            writer.add_scalar('train/log_magnitude', reduced_gen_mag, global_step=iteration)
            writer.add_scalar('train/loss', reduced_gen_loss, global_step=iteration)
            iteration += 1
            if iteration % 100 == 0:
                print(f"Iteration {iteration} : loss {reduced_gen_loss}   mse {reduced_gen_mse}  spectral convergence
{reduced_gen_sc}  log magnitude {reduced_gen_mag}\n")
            if iteration % train_config['checkpoint_it_interval'] == 0:
                save_checkpoint(model, optimizer, iteration, epoch, checkpoint_dir)
                exit = True
                break
        if exit:
            validate(model, mseloss, stft_loss, writer, epoch, iteration)
            break
if __name__=='__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', '--config', type=str, ='JSON file for configuration')
    parser.add_argument('--checkpoint', type=str, default='', help='checkpoint_path')
    args = parser.parse_args()
    with open(args.config) as f:
        data = f.read()
    config = json.loads(data)
    train_config = config["train"]
    global data_config
    data_config = config["data"]
    global model_config
    model_config = config["generator"]
    train(args.checkpoint)
```