Frankfurt University of Applied Sciences

Master Thesis

Anomaly Detection in Machine State Using Machine Learning Algorithm
(A Deep Learning Approach)

Author:
Muhammad Ehsan-Ul-Haq
1098587

Supervisors:
Prof. Dr. Andreas Pech
Prof. Dr. Herbert Nosko

A thesis submitted in fulfillment of the requirements
for the degree of Master of Engineering in Information Technology

in the Department

Fachbereich II - Informatik und Ingenieurwissenschaften

January 15, 2018

Declaration of Authorship

I, *Muhammad Ehsan-Ul-Haq*, declare that this thesis titled, *Anomaly Detection in Machine State Using Machine Learning Algorithm* and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed:

Date:

Abstract

Anomaly detection can be a tedious task, when done manually it can take up lot of man hours and can cost lot of money especially, in environments with highly sophisticated machinery. In order to automate this task, we presented a solution of detecting audio anomalies using a machine learning approach. We observed that the use of acoustic signals originating from the equipment can give us lot of information about machine state. Therefore, we decided to create a learning classifier to effectively distinguish **"normal"** and **"abnormal"** sound signals. We collected the audio clips from equipment in a machine room and used them to learn the spectrum of normal and abnormal sounds.

Due to recent advances in deep learning, we decided to create a deep model with several layers and used a supervised approach to learn the classifier. But, training a deep learning model from scratch requires lot of data and can take up lot of time (many days or weeks). Keeping all these constraints in mind, we decided to use transfer learning approach by taking a pretrained model, removing the last layer, adding a new layer on top, and by forcing it to learn on our new data to classify sounds.

For the sake of this thesis, we worked in collaboration with **Rovema Gmbh**. It is an international company which provides packaging equipment. They have various packaging machines which use auger dosing method to fill the products (e.g. Powder or Coscus) in the packages. Sometimes at high operating speeds, auger starts to scratch the wall of machine and contaminate the food. Generally, they have people who after working many years with these machines can tell when scratching occurs. But, for a new worker it is hard to distinguish between normal and scratching sounds. We analyzed audio data from the machine and presented a solution to this problem by using a deep learning approach.

Finally, we discussed the feasibility of our learning approach and how we can extend it in an industry setting for multiple machines. Main idea behind the approach was to use a separate model for each machine, and automate this task by generating events if certain anomaly occurs in any machine. We also showed how we can relabel data in case of false alarm to support the process of active learning.

Acknowledgment

First of all, I would like to thank my thesis advisors, especially **Prof. Dr. Andreas Pech**, and generally **Prof. Dr. Herbert Nosko** of Engineering Department at Frankfurt University of Applied Sciences for their enormous support. They were always ready to listen to me whenever I ran into trouble or had any questions. Their consistent guidance steered me into right direction, and enabled me to finish my thesis in time.

I would also like to thank **Mr. Kuss** and **Mr. Wenzel** at **Rovema Gmbh** for helping me gather test data and allowing me to perform tests at their equipment. Without their passionate participation and input, the experiment could not have been completed successfully.

Moreover, I also want to acknowledge my colleagues at Frankfurt University of Applied Sciences for proof reading this thesis, and for their valuable comments.

Finally, I want to express my profound gratitude to my father **Muhammad Asghar Ali** and siblings for providing me with their fulfilling support and encouragement throughout my studies. Without their support, this accomplishment would not have been possible. Thank You!

Author,
Muhammad Ehsan-Ul-Haq

Contents

Chapter 1

Introduction

Anomaly detection is one of the very important problems in many application domains. Sometimes in highly sophisticated environments, it becomes quite important to detect any abnormality in machine behavior at an early stage to avoid any further damage to the equipment. Manually performing this task can take lot of effort, and yet one can't guarantee the chance of any critical anomaly going undetected.

Normally, commercial buildings or factories have many machine rooms with different kinds of motors, rotating equipment, mixers, plants, fans and generators. At present, maintenance and health checking occurs only when a complaint is made or several months have passed since last check was made. In latter case, commonly an inspector walks around in the machine room to listen any abnormal sounds. Both of these approaches can cause small problems to eventually become bigger and even costly, since the faults are only found when they have been in effect for a while. For example, a scratching mixer in food packaging industry can seriously contaminate food, and the cost of replacing a damaged motor can go up to thousands of Euros [1].

Audio sensors are normally cheap and easy to use. Also, audio analysis is currently an active area of research [2] and has been applied in range of applications. In this demonstration, we chose to use an architecture which uses acoustic signals, originating from various kind of machines, and used them to train our model. Specifically, we presented a framework that analyzes the sound clips collected from various moving parts of machines in a machines room environment to learn normal and anomalous behavior. We also demonstrated, how a new sound clip can be uploaded, analyzed and how anomalies can be detected and reported.

1.1 Machine health monitoring

Let's consider a rotating machine, for example a mixer or a car engine. These machines produce specific kind of noises, which are somehow related to their rotating speed e.g. putting a mixer to a higher speed produces a high frequency noise. On the other hand, a car engine has more complex build and many vibrating sources inside an engine box contribute to the overall noise. A car driver gets used to these machine sounds during a normal operation. When a car gets older, faults may develop inside the engine. Sometimes, a driver can recognize sudden strange noise among familiar car noises. These sudden noises are called anomalies. After, diagnosing the problem, this may turn out that strange noise was coming from dust or some loosened bolt. At this stage, this may seem harmless, but in future this small problem can lead to potentially a very dangerous

situation. Therefore, machine health monitoring and timely diagnosis at an early stage is very important, and can save lot of trouble in future, or a serious health damage in case of a car accident on a highway [3].

In this thesis, we proposed the solution to a similar problem as mentioned above, using a deep learning algorithm called Convolutional Neural Networks (CNNs) on frequency spectrum of acoustic signals. Algorithm was trained with series of spectral images with labels of normal and abnormal sounds of a certain machine to learn the spectral signature of machine health, under normal and abnormal circumstances. Later, an architecture was proposed to automate the task and generate an alarm or message under abnormal circumstances.

Chapter 2 describes the background information about deep learning models and related work in the area of machine learning with CNNs. Chapter 3 describes the architecture of our learning model and how we can use Transfer Learning to apply the learning from one domain to another. Chapter 4 describes data collection, training and analyzing the experimental results. In the last Chapter 5, we discussed the potential of the technique and future work.

Chapter 2

Background information

2.1 Deep learning

The term **"Deep learning"** refers to large scale artificial neural networks with many layers. The idea of biologically inspired neural networks has been around for several decades now. But, main reason behind recent hype and success is more data and computation power available at our hands. Today, it is one of the fastest growing area in machine learning because of its ability to understand very large and complex data sets. The real breakthrough happened when in 2012, G. E. Hinton and his students managed to beat the previous state of the art prediction systems on most famous datasets: MNIST, TIMIT, CIFAR-10 and ImageNet [4]. And, these are quite mature datasets covering text, speech and image classification. Same year, they also won the ILSVRC-2012 competition by achieving top-5 test error rate of 15.3%, compared to the second best with 26.2% [5].

2.1.1 Motivation

Modern deep learning is a very powerful framework especially for supervised learning tasks. A deep network, by adding more layers and hidden units in each layer can represent functions with increased complexity. Most tasks of mapping an input vector to an output vector, which are very simple for human beings, can be very complex for computers. For example, object detection or telling the difference between a cat and dog can be very simple for us, but it is very complex for machines. Research has shown that, given a sufficiently large dataset and a sufficiently large model, we can accomplish this task at fairly human level accuracy or even better in some cases [6].

Andrew Ng, a Stanford University professor and former founder of Google Brain, which eventually resulted in production of many deep learning technologies, talked why deep learning is taking off at ExtractConf 2015 titled: "Why data scientists should know about deep learning". He commented on why for the old generation learning algorithms performance will plateau and why deep learning is the first-class algorithm, which is scalable and whose performance just keeps increasing as you feed them more data. Figure 2.1 is an extract from Andrew Ng slides.

Apart from scalability, one of the most important advantage of deep learning is its ability to perform feature extraction on their own from data. This is often called feature learning or learning representations. Yoshua Bengio, another pioneer in deep learning has described that deep learning has the ability to learn/discover good representation of features in his paper titled "Deep Learning of Representations for Unsupervised and Transfer Learning". Whereas in traditional machine learning algorithms, many

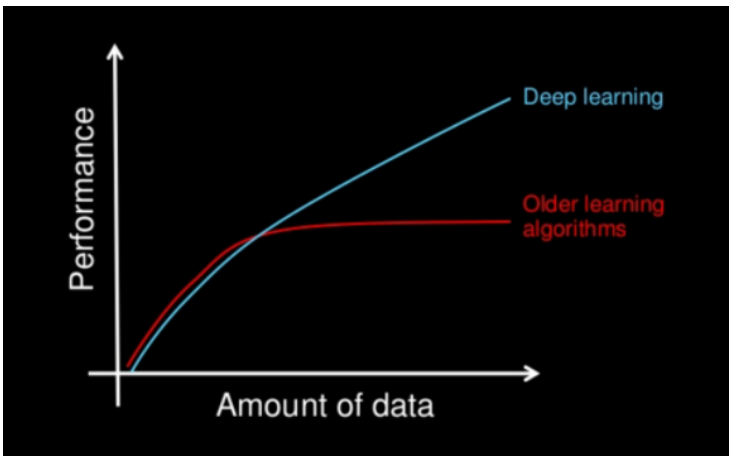


Figure 2.1: Why deep learning? Andrew Ng at ExtractConf 2015

practitioners have solely relied on hand-crafting the representations. [7]

2.2 Deep feedforward networks

Deep feedforward networks are often known as **feed forward neural networks** or **multilayer perceptrons (MLPs)**. The main goal of these networks is to find a function for a classifier, $y = f(x;\theta)$ that maps an input x to an output category y, and learns the value of parameters θ that best approximates the function. The reason it is called feed forward is because the information x flows in forward direction through network defined by f to the output y, and there are no feedback connections where information is fed back to itself. **Convolutional neural network** is one of the examples of feedforward networks [8].

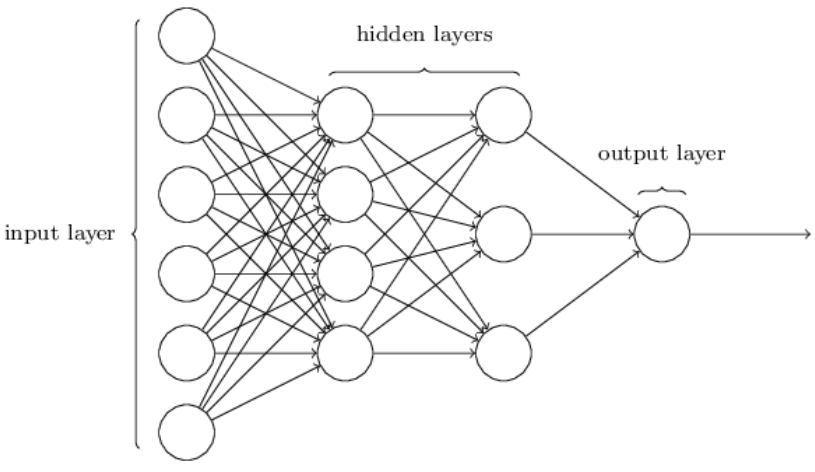


Figure 2.2: A simplified deep feedforward network. [9]

The reason they are called **networks** is because they are built by composing together series of functions in a form of a chain. For instance, we might have three functions f^1, f^2, f^3 to form $f(x) = f^3(f^2(f^1(x)))$, and in this case, f^1 is the first layer of the network, f^2 is second layer and f^3 is the third layer. The total number of layers gives the depth of the network. First layer, which takes x as input is called input layer. Final

layer of the network produces output y, is called output layer. The layers in between are called hidden layers. Each training example x is accompanied by a label y and is passed through network. The learning algorithm must decide how to use hidden layers to produce the output which is as close as possible to y. Figure 2.2 shows the most basic structure of a deep feedforward network [8].

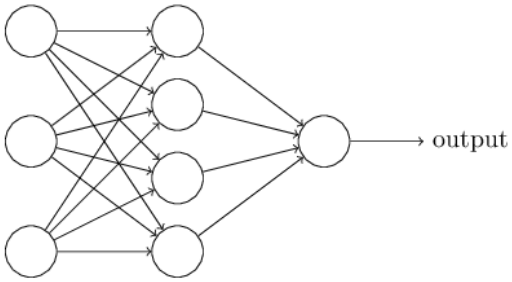


Figure 2.3: A feedforward network with one hidden layer. [9]

As you can see in Figure 2.2, each layer can have one or more nodes/units. Nodes in input layer are called input neurons, nodes in the output layer are called output neuron and similarly, nodes in the hidden layer are called hidden neurons. Each neuron can accept many inputs and returns a single output. Lets consider a simpler architecture with just one hidden layer as show in Figure 2.3. First hidden unit accepts three features as inputs x_1, x_2, x_3 and outputs a value a_1. Let $a_1^{[1]}$ denote the output of first hidden unit in first hidden layer. Layer numbers are zero indexed, that means input layer is layer 0, first hidden layer is layer 1 and output layer is layer 2. Given this mathematical notation, the output of layer 2 is $a_1^{[2]}$. So, we can write input features to first hidden unit as following:

$$x_1 = a_1^{[0]}, x_2 = a_2^{[0]}, x_3 = a_3^{[0]} \tag{2.1}$$

In general form [10], $a_j^{[l]}$ refers to activation/output of j_{th} unit in layer l. $x^{(i)}$ represents i_{th} training example. Note: anything in square brackets refers to layer number and anything in parenthesis refers to training example number. As shown in Figure 2.4, each

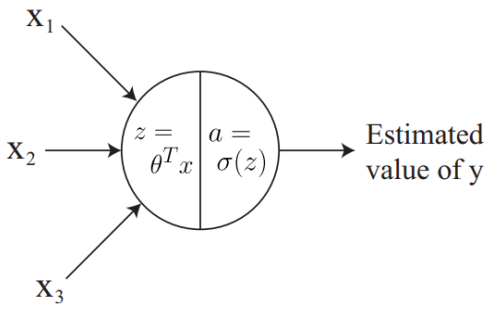


Figure 2.4: Computations in a single neuron [10]

hidden unit performs two computations on its inputs as following:

$$z_1^{[1]} = W_1^{[1]^T} x + b_1^{[1]} \quad \text{and} \quad a_1^{[1]} = g(z_1^{[1]}) \tag{2.2}$$

Here W is a matrix of parameters and W_1 represents the first row of the matrix. The parameters associated with first hidden unit is a vector $W_1^{[1]} \in \mathbb{R}^3$ and a scaler $b_1^{[1]} \in \mathbb{R}$.

And, $g(z_1^{[1]})$ is the activation applied on the output of first hidden unit. Activation function are nonlinear functions and they are applied to add non-linearity to our network so that it can learn any nonlinear mapping of input to output. Following are some of the most famous activation functions.

$$g(z) = \frac{1}{1+e^{-z}} \qquad \text{(sigmoid)} \tag{2.3}$$

$$g(z) = \max(z, 0) \qquad \text{(ReLU)} \tag{2.4}$$

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \qquad \text{(tanh)} \tag{2.5}$$

Similarly, for the second, third and fourth hidden unit in the first hidden layer, computations are as following:

$$\begin{aligned} z_2^{[1]} &= {W_2^{[1]}}^T x + b_2^{[1]} \quad \text{and} \quad a_2^{[1]} = g(z_2^{[1]}) \\ z_3^{[1]} &= {W_3^{[1]}}^T x + b_3^{[1]} \quad \text{and} \quad a_3^{[1]} = g(z_3^{[1]}) \\ z_4^{[1]} &= {W_4^{[1]}}^T x + b_4^{[1]} \quad \text{and} \quad a_4^{[1]} = g(z_4^{[1]}) \end{aligned}$$

Moving to the output layer, it performs following computations

$$z_1^{[2]} = {W_1^{[2]}}^T x + b_1^{[2]} \quad \text{and} \quad a_1^{[2]} = g(z_1^{[2]}) \tag{2.6}$$

Now, we have all the computations of all the nodes in a network, so we can represent the input and output of the hidden layer as a matrix equation for single example as following:

$$\underbrace{\begin{bmatrix} z_1^{[1]} \\ \vdots \\ \vdots \\ z_4^{[1]} \end{bmatrix}}_{z^{[1]} \in \mathbb{R}^{4\times 1}} = \underbrace{\begin{bmatrix} — & {W_1^{[1]}}^T & — \\ — & {W_2^{[1]}}^T & — \\ & \vdots & \\ — & {W_4^{[1]}}^T & — \end{bmatrix}}_{W^{[1]} \in \mathbb{R}^{4\times 3}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{x \in \mathbb{R}^{3\times 1}} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_4^{[1]} \end{bmatrix}}_{b^{[1]} \in \mathbb{R}^{4\times 1}} \tag{2.7}$$

Activation is applied on each element of output vector $z^{[1]}$ as $a^{[1]} = g(z^{[1]})$. Output layer activations and output is computed as following:

$$\underbrace{z^{[2]}}_{1\times 1} = \underbrace{W^{[2]}}_{1\times 4} \underbrace{a^{[1]}}_{4\times 1} + \underbrace{b^{[2]}}_{1\times 1} \quad \text{and} \quad \underbrace{a^{[2]}}_{1\times 1} = g(\underbrace{z^{[2]}}_{1\times 1}) \tag{2.8}$$

2.3 Training

A network is trained on many examples. Let's suppose we had three training examples [10], so we can stack them together as columns in a matrix

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} \tag{2.9}$$

As we stacked the training examples as columns in input matrix X. Therefore, we can combine the inputs and outputs in a single unified formula as following:

$$Z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = W^{[1]} X + b^{[1]} \tag{2.10}$$

$$z^{[2]} = \begin{bmatrix} z^{[2](1)} & z^{[2](2)} & z^{[2](3)} \end{bmatrix} = W^{[2]} Z^{[1]} + b^{[2]} \tag{2.11}$$

Suppose, we had a training set of m labeled examples $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$. Initially, we set the parameters $W^{[1]}, b^{[1]}$ and $W^{[2]}, b^{[2]}$ to small random numbers. Then we compute the output prediction (probability) for each example using **sigmoid** activation $a^{[2](i)}$. Let's denote the activations of output layer $a^{[2]}$ as $\hat{y}$ (predicted value). Now, we can define our **loss function** $\mathcal{L}$ for single example as following [10]:

$$\mathcal{L}(\hat{y}, y) = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}) \right] \tag{2.12}$$

For entire training set $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$, we can define our **cost function** J as following:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \tag{2.13}$$

$$= -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right] \tag{2.14}$$

The main goal of entire training process is to minimize the **cost function** J.

2.3.1 Learning parameters

Before we start training process, it's very important to initialize parameters W and b to some value. We do not initialize them with zero because it causes problem when we try to update them (e.g. all the gradients will be same). So, we initialize them randomly (normally distributed $\mathcal{N}(0, 0.1)$ around zero). Experiments have shown that there is even better way to initialize the parameters instead of random initialization know as **Xavier/He** initialization. For more information read the paper titled:"Understanding the difficulty of training deep feedforward neural networks" by Xavier Glorot and Yoshua Bengio [11].

Once the parameters are initialized, we can start training the network using **gradient descent** algorithm. The loss function $\mathcal{L}$ produces a scalar value for single example and the cost function $J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}^{(i)}$ is defined as the average of loss values for all training examples m. For any layer l we update them as following [10]:

$$W^{[l]} = W^{[l]} - \alpha \frac{\partial J}{\partial W^{[l]}} \tag{2.15}$$

$$b^{[l]} = b^{[l]} - \alpha \frac{\partial J}{\partial b^{[l]}} \tag{2.16}$$

Here α is the learning rate. It is very important to tune the value of α during training process. It is a common practice start it with 0.1 and gradually decrease it over time. Paper "An empirical study of learning rates in deep neural networks for speech recognition" published by Google, talks about it in greater details [12].

2.3.2 Optimization

The update rule defined by **gradient descent** in Equation 2.15 and Equation 2.16 is not practically feasible. Because, in one forward pass it needs the loss $\mathcal{L}$ values for all training examples to compute a cost J, and most of the times datasets are so large that it's impossible to hold them in RAM. The other solution is to use **stochastic gradient**

descent (SGD) algorithm. Instead of using cost value, we use loss values for each example to update the parameters in forward and backward pass as following:

$$W^{[l]} = W^{[l]} - \alpha \frac{\partial \mathcal{L}}{\partial W^{[l]}} \tag{2.17}$$

$$b^{[l]} = b^{[l]} - \alpha \frac{\partial \mathcal{L}}{\partial b^{[l]}} \tag{2.18}$$

SGD can be very fast because it starts updating parameters after every example. The only problem associated with it is that it can be very noisy, since it use loss values which can vary largely for two consecutive examples. So, we need to find a middle ground between GD and SGD. In practice and research, we use a technique called **mini-batch gradient descent** algorithm. It's a compromise between GD and SGD. In minibatch gradient descent algorithm, the cost function function J_{mb} is defined as following [10]:

$$J_{mb} = \frac{1}{B} \sum_{i=1}^{B} \mathcal{L}^{(i)} \tag{2.19}$$

Where B is the number of examples in a minibatch. As, minibatch gradient descent is an optimization technique, therefore it comes with few challenges too. One of the key challenge is selecting the proper value of learning rate α. Too small learning rate will slow the training process and too large can hinder the convergence and can cause the loss function to fluctuate around the minimum or even diverge. Few other techniques e.g. **momentum, Adagrad, RMSprop, Adam** have been built to improve the optimization by using adaptive learning rates. Paper "An overview of gradient descent optimization algorithms" by Sebastian Ruder provides a comparison of these techniques [13].

2.3.3 Regularization

Sometimes, training model achieves high accuracy on training set e.g. 96% and a low accuracy 64% on test set. The reason behind this is that the model is over-fitting the training set and can't generalize very well on the test set. One of the solutions to this problem is to collect more training data or reduce the number of layers. If you can't do either of these then there is a technique called regularization. The main idea behind regularization is to penalize the complex models for their complexity by adding a regulation term. L_2 regularization is one the most common technique. In L_2 regularization, another term is added to the cost function as following [10]:

$$J_{L2} = J + \frac{\lambda}{2} \|W\|^2 \tag{2.20}$$

$$= J + \frac{\lambda}{2} \sum_{ij} |W|^2 \tag{2.21}$$

$$= J + \frac{\lambda}{2} W^T W \tag{2.22}$$

Where J is as standard cost function and λ is an arbitrary value indicating the intensity of regularization. W represents all the weights in a network. So, with L_2 regularization update rule for equation Equation 2.15 and Equation 2.16 becomes:

$$W = W - \alpha \frac{\partial J}{\partial W} - \alpha \frac{\lambda}{2} \frac{\partial W^T W}{\partial W} \tag{2.23}$$

$$= (1 - \alpha\lambda) W - \alpha \frac{\partial J}{\partial W} \tag{2.24}$$

Equation 2.24 shows that every update will include some penalization depending on W. The penalization increases the overall cost J, and makes the individual parameters to be small in magnitude, which in return reduces the over-fitting.

2.3.4 Batch normalization

Batch normalization is a recent technique presented by Sergey Ioffe and Christian Szegedy at Google [14], which allows us to use higher learning rate (faster training) and to be less careful about parameter initialization when using minibatch gradient descent algorithm during training. As we know that deep neural networks learn better when training data is normalized (e.g. unit variance, zero mean), which make the data more comparable across features. But, when this data flows through layers, due to changing values of parameters W and b, it may become either too small or too big, which is a problem, authors of the paper refer to as **"internal covariant shift"** [14]. So instead of doing this normalization just at the beginning, we do it at each layer.

Let's consider the output $z_i^{[l]}$ as described in Equation 2.2 of an arbitrary node i in layer l of a deep network. For a specific minibatch B with n examples, we have $(z_i^{[l](1)}, \ldots, z_i^{[l](n)})$ outputs. First, we calculate the mean and variance for outputs $z_i^{[l]}$ of each minibatch as following:

$$\mu_B = \frac{1}{n}\sum_{j=1}^{n} z_i^{[l](j)} \tag{2.25}$$

$$\sigma_B^2 = \frac{1}{n}\sum_{j=1}^{n}(z_i^{[l](j)} - \mu_B)^2 \tag{2.26}$$

And, then we normalized $z_i^{[l]}$ by subtracting the mean and dividing by the variance as following:

$$\widehat{z}_i^{[l]} = \frac{z_i^{[l]} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.27}$$

Then,we re-shift and re-scale them by a new set of parameters β and γ. These new parameters are initialized with zero mean and unit variance $\beta = (0_{(1)}, \ldots, 0_{(i)})$ and $\gamma = (1_{(1)}, \ldots, 1_{(i)})$, where i is number of nodes in a particular layer. They perform $y_i^{[l]} = \gamma\widehat{z}_i^{[l]} + \beta$ re-scaling and re-shifting on the output. The main purpose of β and γ is to undo normalization for particular features if necessary. They are learned similarly as W described in equation Equation 2.15

$$\beta^{[l]} = \beta^{[l]} - \alpha\frac{\partial J_{mb}}{\partial \beta^{[l]}} \tag{2.28}$$

$$\gamma^{[l]} = \gamma^{[l]} - \alpha\frac{\partial J_{mb}}{\partial \gamma^{[l]}} \tag{2.29}$$

Finally, the activation is applied $\widehat{a}_i^{[l]} = g(y_i^{[l]})$. Also, when using batch normalization, we can remove the bias b (set them to zero for all layers) because when we normalize the outputs, it nullifies its effect. So, there is no point of using it.

2.4 Convolutional Neural Networks (CNNs)

Convolutional neural networks CNNs are one of the most impactful variant of neural networks and have played an important role in the history of deep learning. They were one of the first neural networks to solve important commercial problems of reading handwritten digits in 90s and still are at the forefront of computer vision and deep learning. The real breakthrough happened in 2012, when they beaten all state of the art classifiers in an ImageNet object recognition challenge ILSVRC-2012 with lowest top-5 error rate of 15.3% [5]. Since then, research into CNNs has processed so rapidly that many architectures of CNNs with lot of improvements had consistently been proposed.

2.4.1 Motivation

Let's recall the logistic regression, the parameter vector $\theta = (\theta_1, \ldots, \theta_n)$ as shown in Figure 2.4 must have the same number of elements as the input vector $x = (x_1, \ldots, x_n)$. Hence, θ_1 always looks at the pixels on the top left corner of the image. On the other hand, the soccer ball can appear at any place in the image and doesn't have to appear always at the center. And, it might be possible that during training time network was never trained with an image containing soccer ball at top left corner. Therefore, at test time if a soccer ball appears at top left, our model will most likely predict "no soccer ball". This problem leads us to the "convolutional neural networks".

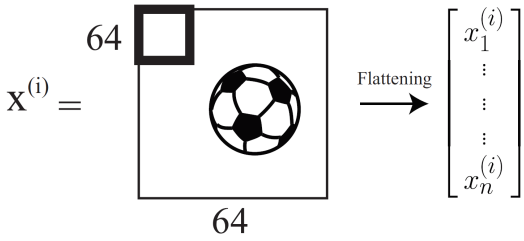


Figure 2.5: Soccer ball problem in logistic regression. [10]

Consider Figure 2.5, and suppose θ is no longer a vector but a matrix of 4×4 pixels. We take this matrix of parameters θ and slide it all over the image. We then compute element wise product of θ and sliding windows x of image. Then we collapse the matrix by summing all the terms of element wise product into a single scalar value as following:

$$a = \sum_{i=1}^{4} \sum_{j=1}^{4} \theta_{ij} x_{ij} \tag{2.30}$$

Figure 2.6 shows different element wise products at different locations of image. When

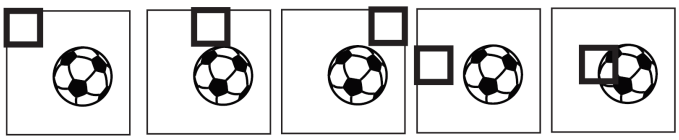


Figure 2.6: Element wise multiplication using sliding window on a soccer ball image. [10]

we reach at the end of the image, the parameters θ have seen all the pixels of image. Hence, θ_1 is no longer bound to top left corner of the images. Now if soccer ball appears at any place of the image, our network will successfully detect it [10].

2.5 Basic components of CNN

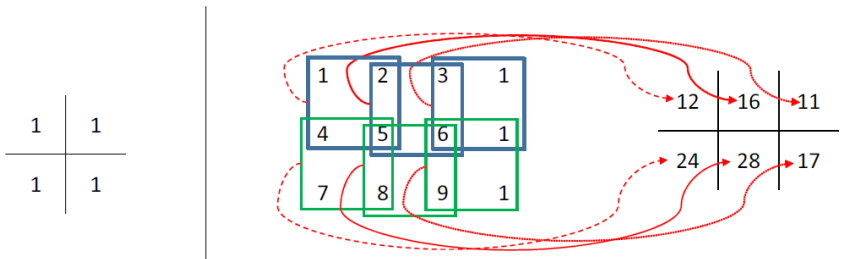
A CNN consists of series of processes and usually takes a tensor of order 3 (e.g. in case of image R, G, B channels) as input. Each of these processing steps are called layers. Either of these processing steps could be a convolutional layer, pooling layer, activation (ReLU) layer or fully connected layer etc.

Before we start discussing each of these layers, lets first define input and output notations. Let's consider the l_{th} layer, with an input of an order 3 tesnsor x^l with $x^l \in \mathbb{R}^{H^l \times W^l \times D^l}$. Thus, in order to locate x^l we need a triplet index set (i^l, j^l, d^l). This triplet (i^l, j^l, k^l) refers to one element in tensor x^l, which is spatially located at (i^l, j^l) in a d^l_{th} channel. In practice, CNN uses mini batches. In that case, x^l becomes a tensor of order 4 $\mathbb{R}^{H^l \times W^l \times D^l \times N}$, where N is the size of minibatch. For the sake of simplicity we use $N = 1$ and all the indexings are zero based e.g. $0 \leq i^l < H^l, 0 \leq j^l < W^l$ and $0 \leq d^l < D^l$.

After passing through each layer, x^l will transform to an output y, which is then fed into next layer as an input. So, we can represent y as x^{l+1}. And, the output $H^{l+1} \times W^{l+1} \times D^{l+1}$ is represented by a triplet $(i^{l+1}, j^{l+1}, d^{l+1})$ with zero indexing $0 \leq i^{l+1} < H^{l+1}, 0 \leq j^{l+1} < W^{l+1}$ and $0 \leq d^{l+1} < D^{l+1}$ [15].

Convolution layer

Convolution layer is one of the most important layers in CNN. Normally, it is composed of series of filters or kernels. When applied on input these filters can learn from low to high level features of training set. Let's suppose we have a convolution filter of size 2×2 and an input image of size 3×4 as shown in following figure.



(a) A 2×2 kernel

(b) The convolution input and output

Figure 2.7: Illustration of convolution operation [15]

If we overlap the filter on top of an image, we can calculate the products between numbers in the same location, and we get a single number by summing all these products. We then move this filter all over the image, sliding element (pixel) by element (pixel), from left to right and top to bottom, until it reaches the bottom right corner of the image. For an order 3 tensor of size $(H^l \times W^l \times D^l)$, a convolution filter is also of order 3 $(H \times W \times D^l)$. We overlap this filter on top of input tensor and compute the products on all spacial location in all three channels and sum the results to get a single scalar value. Normally, each convolution layer has many filters f and we repeat the same process for every filter. Therefor, output is also a tensor of size $(H^l \times W^l \times N)$, where N is the number of filters f in a convolution layer. **Stride** represents the number of pixels we jump when we move the filter from left to right and top to the bottom of the image. Stride $s = 1$ represents that we skip no pixel and stride $s = 2$ means we skip $s - 1$ pixels each time we move the filter. As shown in Figure 2.7, spacial dimension of output is smaller than the input and

sometimes we want to preserve the dimension throughout the network. This can simply be achieved by adding **padding** of zeros. If we have stride of 1 and filter size $K =$ (height H or width W), then we must add a padding of $(\frac{K-1}{2})$ to preserve the spacial dimension. Similarly, we can compute the size of output as following:

$$outputsize = \frac{inputsize - filtersize + 2 \times padding}{stride} + 1 \tag{2.31}$$

Activation (ReLU) layer

Activation (ReLU) is also one of the important layers in CNN. The main purpose of this layer is to increase non-linearity in CNN. For example, the information in an image (e.g picture of a cat or dog) is a highly non linear mapping of pixel value to a label. And, ReLU function is a simple yet non linear function. It can be defined as following.

$$y_{i,j,d} = max(0, x^l_{i,j,d}) \tag{2.32}$$

Where, x^l is an input and y is an output. A ReLu layer does not change the size of the input. Therefore, x^l and y share the same size. Figure 2.8 shows the ReLu function.

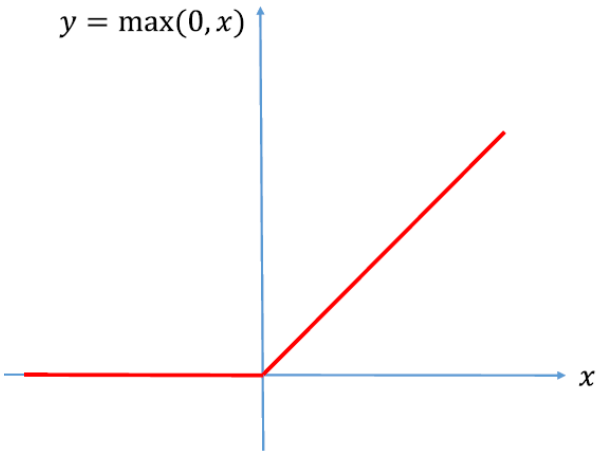


Figure 2.8: The ReLU function [15]

There are other activation functions e.g Sigmoid (Equation 2.3) or tanh (Equation 2.5). Experiments have shown that ReLU works much better than Sigmoid or tanh almost all the time and helps reduce vanishing gradient problem significantly, which make training lot more faster and easier.

Pooling and dropout layer

After ReLU layer, it is a common practice to apply a pooling layer, sometimes also referred to downsampling layer. It takes an input volume of parameters and reduces it to a smaller volume, known as "dimensionality reduction". Which in result reduces the number of computations and over fitting. The most famous types of pooling are "max pooling" or "average pooling". It basically takes a filter of size $K \times K$ (normally 2×2) and a stride of same length. It then applies it to input volume and outputs a maximum number or the average of all the parameters in every sub-region.

To gain extra regularization effect, pooling is sometimes combined with dropout. Dropout

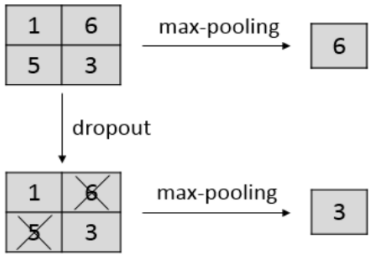


Figure 2.9: An illustration of pooling operation with and without dropout. [16]

operation simply neglects (drops) some parameters at random by setting them to zero. Sometimes, a trained model does really well e.g 100% or 98% on training set but only 50% on test set. This problem is referred as over fitting. Both pooling and dropout make sure that a model does not over fit the training data. Paper "A Simple Way to Prevent Neural Networks from Overfitting" by G.Hinton talks about the problem of over fitting in greater detail.

Dense or Fully connected layer

A dense or fully connected layer is often implemented at the end of CNN. It refers to a layer whose each output element x^{l+1} (or y) requires all the elements in the input x^l. This kind of layer is useful in deep CNN models e.g after many convolution, ReLU and pooling layers, the output of current layer contains distributed representations of an input image. So, in order to combine all these representations into single unified result, a fully connected layer is used.

2.6 Famous CNN architectures

Convolutional neural networks have evolved over time and lot of improvements have been made. They are one of the key models built by reading the insights of a human brain and they are one of the first deep models to work well. Following are few examples of successful CNN architectures.

LeNet

LeNet-5 is one of the first most successful CNN architectures proposed by Yann LeCun (1998) in his paper "Gradient Based Learning Applied to Document Recognition". The main purpose of the model was to recognize handwritten digits. It was trained on MNIST dataset. Which contained 70,000 hand written (60,000 training and 10.000 test) digit examples. Figure 2.10 shows architecture of the network.

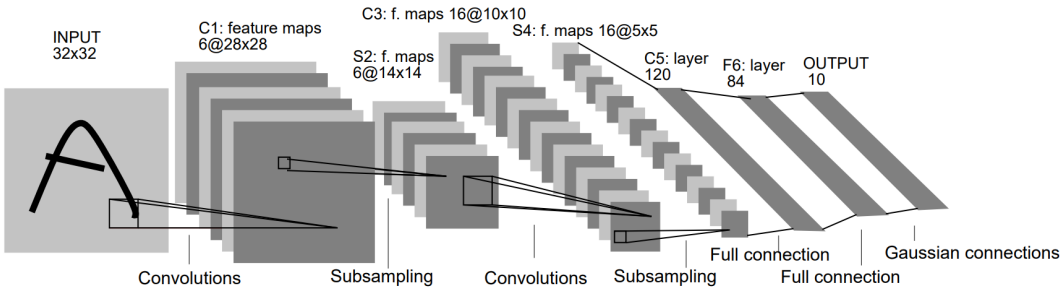


Figure 2.10: Architecture of LeNet-5, a convolutional neural network for handwritten digits recognition. [17]

LeNet-5 is comprised of an input and seven other layers. Input is comprised of 32×32 pixels with normalized values between -0.1 (white) and 1.175 (black). Layer are labeled as Cx, Sx and Fx for convolution, sub-sampling and fully connected respectively. Layer C1 is convolution layer with 6 filters of size 5×5 with stride of 1. Layer S2 is a sub-sampling layer of size 14×14 with 6 filters of size 2×2. Sub-sampling is done by adding 4 units under receptive field, multiplying with trainable coefficient, adding bias and then applying a sigmoid activation function. Layer C3 is again a convolution layer of 10×10 with 16 filters of size 5×5. Then S4, a sub-sampling layer with 2×2 filter. C5 is a convolution layer with 120 units and each unit is connected with 5×5 feature map in layer S4. F6 is a fully connected layer with 84 units and each unit is connected with 120 units in C5. Output layer has ten units to detect hand written digits $(0, \ldots, 9)$ [17].

AlexNet

AlexNet was created by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton. They also presented a paper "ImageNet Classification with Deep Convolutional Neural Networks" that won ILSVRC 2012 competition. CNN shocked computer vision community for the first time with top five test error rate of 15.4%, with the next best entry that achieved 26.2%.

The architecture of AlexNet was relatively simple as compared with modern state of the art. It was composed of five convolution, max pooling and dropout layers, and three fully connected layers. Output layer was used to classify 1000 ImageNet categories. Moreover, it was trained on ImageNet data with two GTX 580 GPUs for six days, which contained

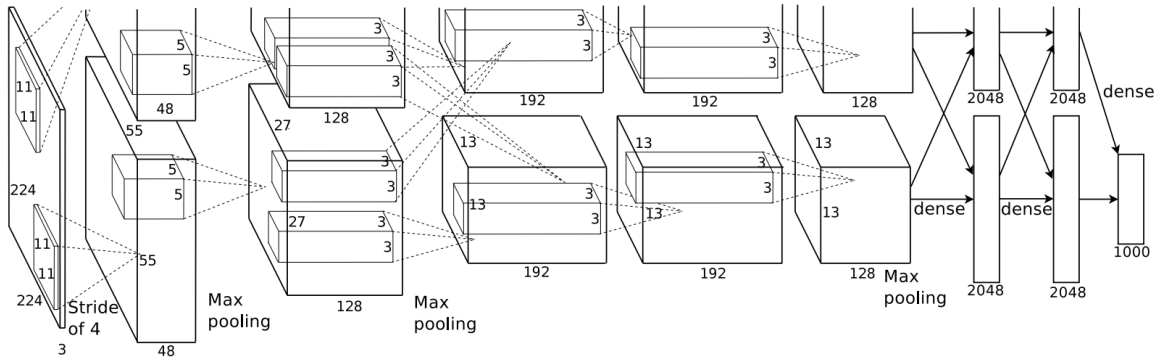


Figure 2.11: Architecture of AlexNet, a convolutional neural network for ImageNet classification. Training process was split on two parallel GPUs because of computational expensiveness. [5]

over 15 million images with 22,000 categories. ReLU was used as activation function and data augmentation techniques (image translation, patch extraction and horizontal reflection) were used. Dropout layers were used to combat over fitting and back prop (stochastic gradient descent) with momentum and weight decay was used for training.

GoogleNet

GoogleNet was a 22 layer deep model presented by Google in a paper "Going deeper with convolutions". GoogleNet was winner of ILSVRC 2014 with a top 5 error rate of 6.7%. It was the first CNN model that completely discarded the idea of simplicity. Figure 2.12 shows overall architecture of GoogleNet.

They presented the idea of inception module with parallel operations as shown in

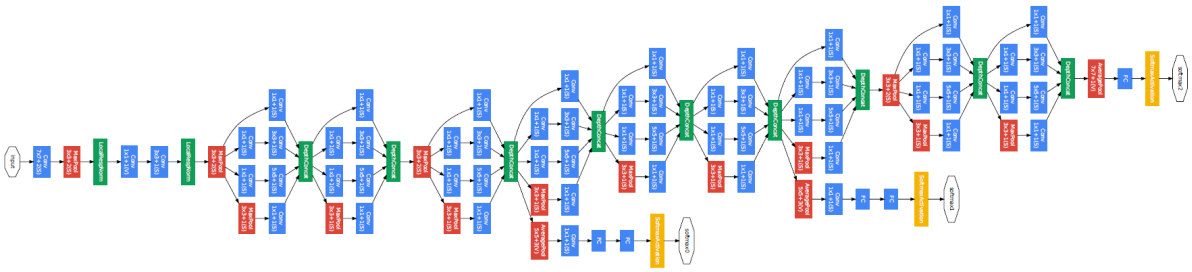


Figure 2.12: Architecture of GoogleNet with all inception modules. [18]

Figure 2.13. The green box is the input and you have to make choice whether to have convolution or pooling operation with various filter sizes. Apparently, it turned out in later architectures that this was a naive idea, which lead to too many outputs. So, they addressed this issue by adding 1×1 convolutions before 3×3 and 5×5 as a dimensionality reduction technique.

They used 9 inception modules in overall architecture. They did not use any fully connected layer, instead they used an average pool from $7 \times 7 \times 1024$ to $1 \times 1 \times 1024$. Overall parameters in GoogleNet were 12 times less than AlexNet. Lastly, model was trained on few high end GPUs for one week [18].

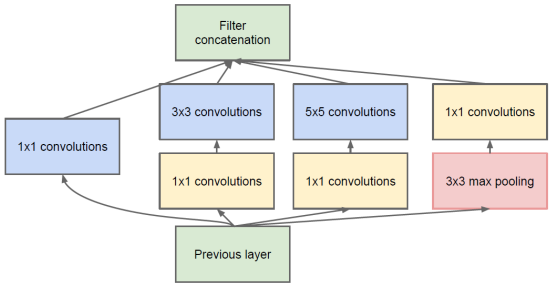


Figure 2.13: Inception module with dimensionality reduction. [18]

ResNet

ResNet was an ultral deep architecture presented by Microsoft in paper "Deep Residual Learning for Image Recognition". It was a simple 152 layer architecture, that set a new record in the areas of classification detection and localization. ResNet also won ILSVRC 2015 with an incredible error rate of 3.6%.

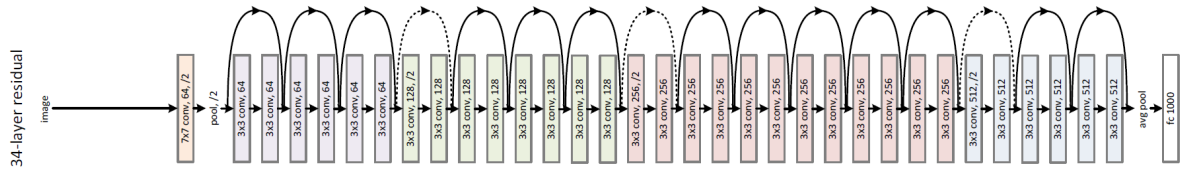


Figure 2.14: Architecture of ResNet-34 model with residual blocks. [6]

They presented the idea of a residual block as shown in Figure 2.15. The main idea behind it was if you have input x and goes through convolution-relu-convolution series, in the end you will have $F(x)$. This result is then added to original input x, that gives us $H(x) = F(x) + x$. So, this mini module computes a slight change to the original input to get slightly altered representation. The authors of the paper believe that it is easier to optimize residual mapping if compared to original mapping.

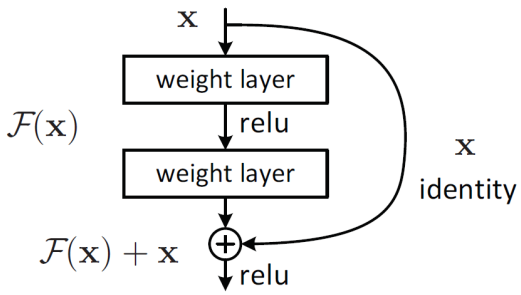


Figure 2.15: Residual block of ResNet model. [6]

The model consisted of series of residual blocks. Authors also observed that naive increase in number of layers resulted in higher training and test error because of over-fitting. It took 8 GPUs and two to three weeks to train 152 layer model [6].

2.7 Proposed method

Until now, we have seen how deep learning has evolved over time, and how various models of convolutional neural networks have helped us solve the hardest problems of computer vision, which once deemed impossible through traditional techniques. Therefore, we decided to choose the same technique to solve our problem of anomaly detection for machine health monitoring. We decided to choose ResNet model and to train it on visual representations of audio to detect the anomaly in machine state. But there is one problem that, deep ResNet model requires a lot of training data and computation power. In order to cope the data scarcity and computation problem, we decided to use following optimization techniques.

Transfer learning

Transfer learning is a technique of taking a pretrained model with the weights and biases trained on a large dataset and fine tuning the model with your own dataset. We based our intuition on two papers "How transferable are features in deep neural networks?" by Yoshua Bengio and "A Deep Convolution Activation Feature for Generic Visual Recognition" by Jeff Donahue. Both of these papers discuss the idea and extent of transferring learning from one domain to another.

The main idea behind the technique is that pretrained model will act as feature extractor and you will remove the last layer of the network and replace it with your own layer depending on your problem space. Then you freeze all the parameters of other layers and retrain the model on your new dataset. The reason this technique works is that in deep model initial layers extract very low-level features. If a deep model is trained on ImageNet dataset with 14 million images and over 1000 classes, then we know that initial layers will always detect extremely small features like edges and curves. So, rather training the whole network with random weight initialization, we can use the weights of a pretrained model and focus on the layers which are higher up in the order.

Data augmentation

Data augmentation is an interesting technique to cope with data scarcity problem. It is a technique of altering data in way that it does not affect true mapping. For example, if you flip the image of a cat from left to right, it is still a cat. Similarly, there are many ways to artificially expand the data set. Most popular augmentation techniques are horizontal and vertical flips, rotation, jitter, translation or random crops. By simply applying these techniques you can expand your data set 2-3 times [19].

Chapter 3

Model architecture and analysis

As we saw previously, many architectures of convolutional neural networks (CNN) have been proposed and improved in past several years. We also observed that performance of CNN improves as we go deeper. Therefore, we decided to choose ResNet architecture for anomaly detection problem. ResNet was proposed by Microsoft Research Asia MSRA team in 2015. They published their results in a break through paper called "Deep Residual Learing for Image Recognition", where for the first time in history, their model surpassed a human level performance on ImageNet with an incredible error rate of 3.6%. They proposed the idea of residual blocks and suggested that their architecture makes deeper nets a lot easier to optimize. They also hypothesized that stacking more layers in residual way increases the performance, which was not possible before with plain net architecture.

3.1 Deep residual learning

Deep residual learning is an idea of stacking layers in a fashion of residual blocks in convolutional neural networks. A residual block consists of two or more convolution layers with shortcut connections. While in a plain architecture layers were simply stacked in serial manner without any shortcut links. Previously, in plain architecture it was observed that as you stack more layers, it become more difficult to optimize and train the network. The real problem with deeper nets was vanishing or exploding gradients. Though, most part of that problem was addressed by normalizing input and adding intermediate normalization layers. After normalization, networks converged easily during training time but their performance degraded rapidly during test time as shown in Figure 3.1.

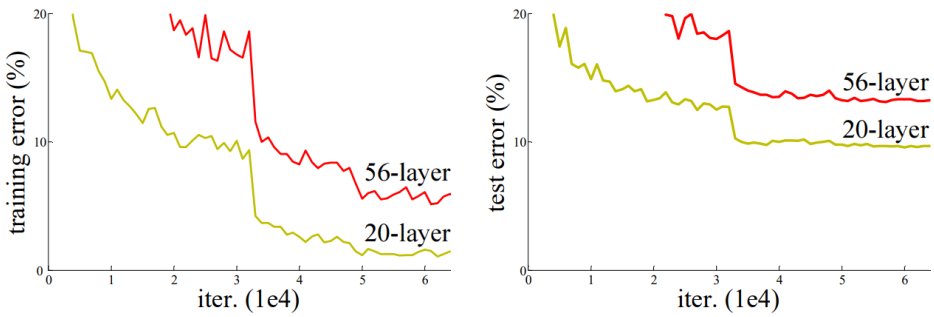


Figure 3.1: Training (left) and test (right) error for "plain net" of 20 and 56 layers on CIFAR-10 dataset [6]

Experiments indicated that this degradation in performance was surprisingly not because

of over fitting. They observed that if we add identity mappings (shortcut connections) in the construction of network, deeper model no longer produces higher training error than its counterpart shallower model. Following Figure 3.2 shows initially proposed architecture of a residual block.

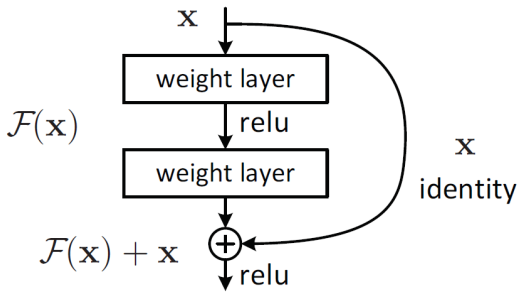


Figure 3.2: Initially proposed architecture of a residual block of ResNet model. [6]

Here, let's denote the mapping of whole residual block with $\mathcal{H}(x)$ and original mapping of underlying two nonlinear layers with $\mathcal{F}(x)$, and x represents the identity mapping of input of residual block. It was observed that it is easier to optimize residual mapping $\mathcal{H}(x) := \mathcal{F}(x) + x$ than the original underlying mapping $\mathcal{F}(x)$. Moreover, identity shortcuts add no extra parameters and hence, model can be easily trained by backpropagation with SGD at no extra computation cost.

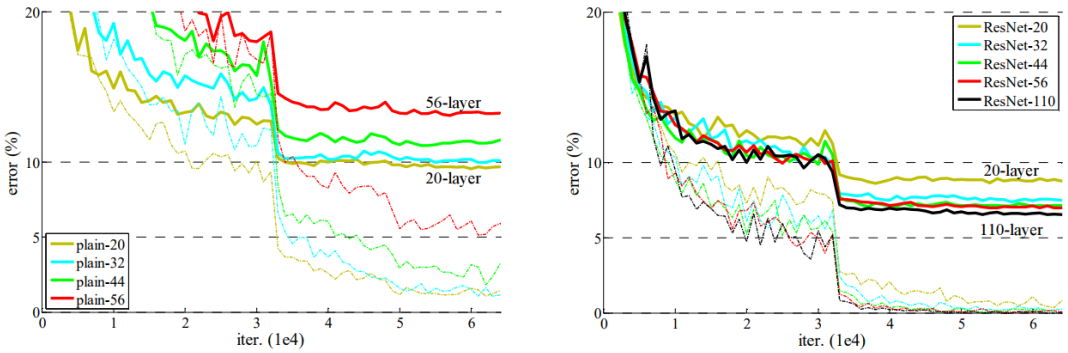


Figure 3.3: Plain net (left) and ResNet (right), training (doted line) and test (solid line) error results on CIFAR-10 dataset. [6]

In [20], Yoshua Bengio also argues, how a deeper model should generalize better than shallower model. After adapting the network to the architecture presented in Figure 3.2, they observed that deeper ResNet models perform better than their counterpart shallower models. Figure 3.3 shows the training and test error results of both plain and ResNet models on CIFAR-10 dataset [6].

3.2 Identity mappings

As presented in Figure 3.2, residual block consists of two or more convolution layers and an identity mapping of input x. In general form, we can represent output y for arbitrary number i of intermediate layers as following:

$$y = \mathcal{F}(x, W_i) + x \tag{3.1}$$

Here, function $\mathcal{F}(x, W_i)$ represents residual mapping to be learned by underlying nonlinear layers. Output y for Figure 3.2 can be represented by $y = W_2\sigma(W_1x) + x$. Second non-linearity is applied after addition $\mathcal{F} + x$ (i.e $\sigma(y)$). There is one problem with this architecture. For an ultra-deep model with 1202 layers, they observed that it no longer performs better than its counterpart shallower model with 110 layers as shown in following Figure 3.4.

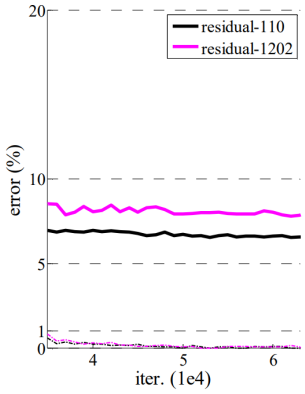


Figure 3.4: ResNet-1202 (test-error 7.9%) vs ResNet-110 (test-error 6.4%) result on CIFAR-10 dataset. [6]

In 2016, MSRA suggested a new modification in residual blocks in their paper "Identity mappings in Deep Residual Networks". They reported an improved error rate of 4.62% for 1001-layer ResNet on CIFAR-10 data set, which with the previous architecture was 7.61%. Similar improvements were also reported on CIFAR-100 and ImageNet datasets.

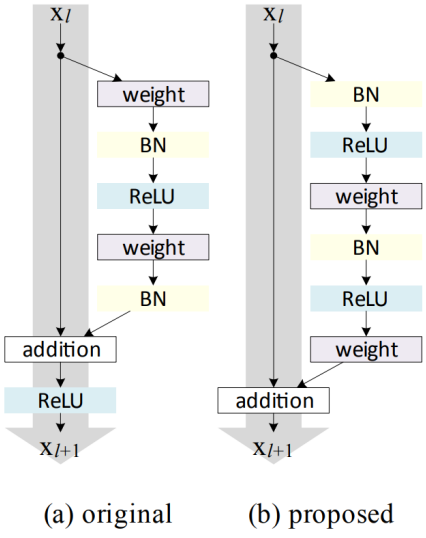


Figure 3.5: Original residual block (a) vs proposed (b) residual block. [21]

By doing various modifications in original architecture, they observed that if we create

a direct path for propagation of information, not only within a residual block but the entire network, it no longer overfits. By choosing the modification proposed in Figure 3.5 (b), they observed that model not only achieves the fastest error reduction but also the lowest training loss in fewer iterations.

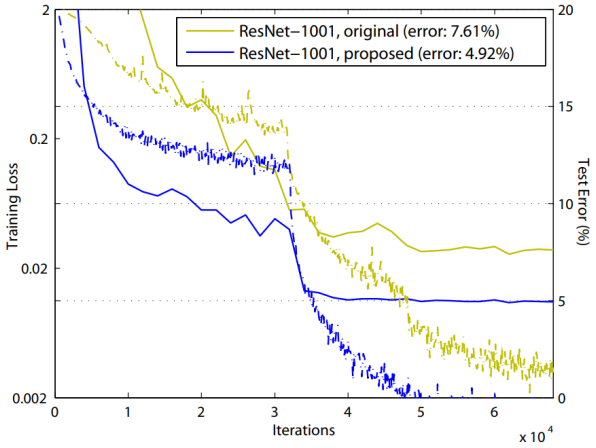


Figure 3.6: Training (doted-line) and test (solid-line) error results for 1001-layer ResNet original(a) vs proposed (b) model. [21]

As shown in Figure 3.5(b), the modification they suggested was removal of activation after addition. Instead they proposed, pre-activation (Batch normalization BN, ReLU) of weight layers. With this new modification they also reported improved results on ImageNet with 200-layer model, which used to overfit with initially proposed (Figure 3.5(a)) residual block architecture. This new modification suggests the importance of identity mappings, and that there is much room to exploit in the depth of the network. Figure 3.6 shows the training and test results of ultra deep ResNet model with 1001 layers for both architectures, and you can see how new architecture achieves low error and loss values for fewer iterations than its counterpart.

By analyzing the above results, we can conclude that pre-activation had two-fold impact. First, it helped to ease the optimization. This effect is very obvious if we look at the results presented in Figure 3.6. Both loss and error curves reduce much faster and in less number of iterations as compared to their counterpart. It can also be observed that effect of ReLU after addition in post-activation (original architecture) is not much when we have lesser layers (e.g. 164) and training curves just suffers initially but goes back into health state soon because of training weights are adjusted quickly to a status where they are more frequently above zero. But when layers increase (e.g. 1000), its effect increases because weights are more frequently less than zero. Second, it had slight regularization effect and that was because of batch normalization. In original architecture (Figure 3.5(a)) after batch normalization we add it to the shortcut and then this merged signal is activated and fed to the next residual block. Therefore, the input of each residual block is non-normalized. But, in proposed architecture (Figure 3.5(b)) all the inputs are normalized first [6][21].

3.3 Network architecture

As we discussed earlier, ResNet is created by simply adding the identity shortcut connections in plain net architecture. These identity shortcuts help improve the information flow and they can be directly applied when input and output are of the same dimensions. Figure 3.7 shows the architecture of a 34-layer ResNet model in comparison with 34-layer plain and VGG-19 [22] network. Here, solid lines represent direct connection (identity mapping) when input-output dimensions are same. While, dotted lines represent the connection (identity mapping) where input-output dimensions are not same. In first case, shortcut performs mapping with no extra paddings, but in second case linear projection $y = \mathcal{F}(x, W_i) + W_s x$ is performed with projection matrix W_s to match dimensions.

Despite being a deeper network, it still has less number of operations as compared to famous VGG-19 [22] network. As shown in Figure 3.7, you can see that VGG network has 19 layers and it has 19.6 billion (floating point operations) FLOPs, while plain net and ResNet both have same number of 3.6 billion FLOPs. Moreover, you can also see that shortcut connections add no extra parameters to the network. Table 3.1 shows the architecture of five ResNet models with total number of FLOPs per each network and you can see that 152-layer model has even less number of FLOPs than VGG-19 architecture.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 3.1: ResNet-18,34,50,101,152 layer architectures for ImageNet with number of blocks stacked. Sub-sampling is performed on each conv1, conv2, conv3, conv4 and conv5 blocks. [6]

All models described in Table 3.1 generally follow same design principles. Only difference is, ResNet-18,34 contain residual blocks of two convolution layers with filter size of 3×3, while ResNet-50,101,152 contain residual blocks of three convolution layers with filters of $1 \times 1, 3 \times 3$ and 1×1 respectively. Whenever output size halves, we double the number of filters to preserve time complexity per layer. Input dimension is 224×224 and output is a fully connected layer of size 1000. Every convolution layer is preceded with batch normalization BN and ReLU activation. There is one max pool at the start and one max pool at the end but they are not counted as separate layers [6].

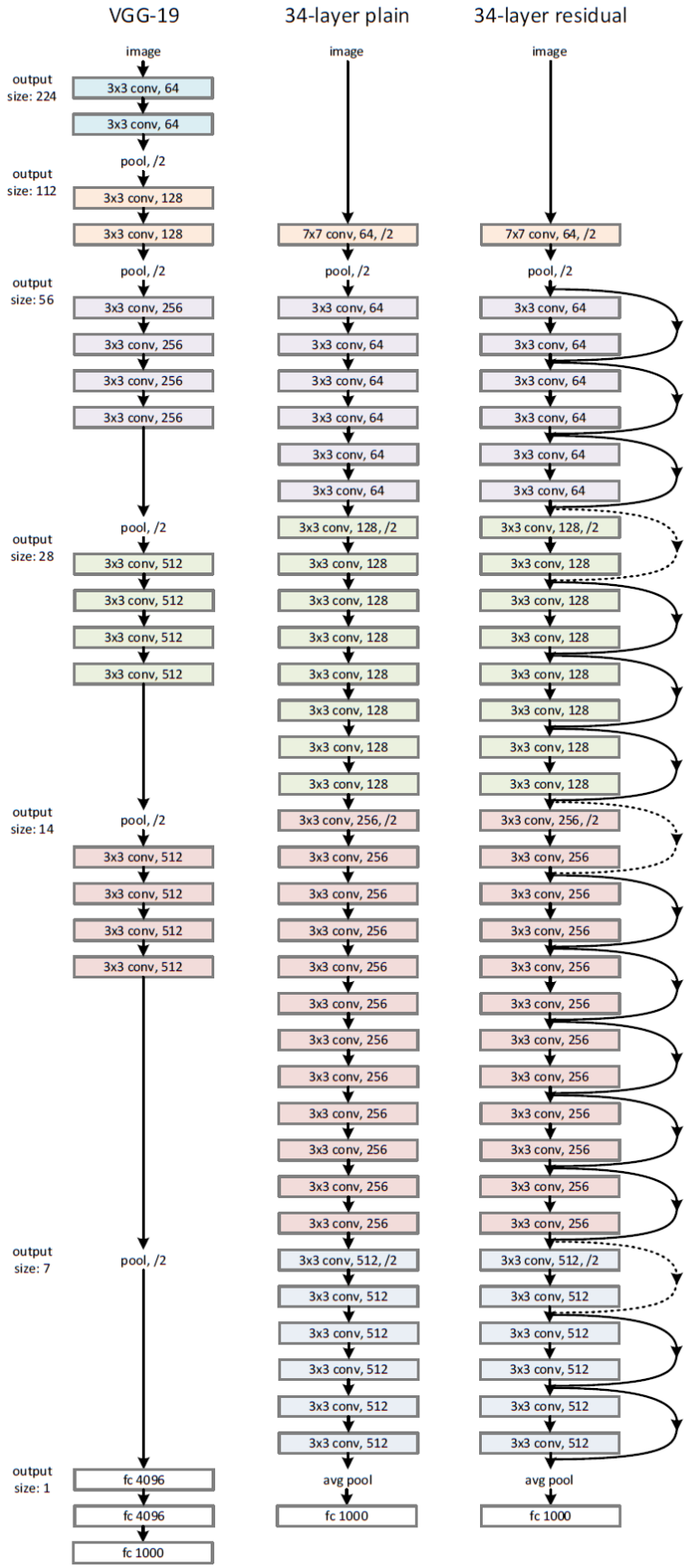


Figure 3.7: Comparison of VGG-19 (19.6 billion FLOPs), PlainNet-34 and ResNet-34 (3.6 billion FLOPs) architectures for ImageNet [6]

3.4 Implementation and training

For the sake of our project, and because of limited data, we decided to choose ResNet-18,34 models only. Both of these models were trained on ImageNet with 1.2 million training, 50,000 validation and 100,000 test images and results were measure by top 1/5 error rates. They used weight decay $\lambda = 0.0001$ and momentum $\beta = 0.9$ for training. Weight initialization was done as in [23] and batch normalization BN as in [14]. The models were trained on 8 GPUs in parallel for about 3 weeks with minibatch size of 256, divided into 32 for each GPU. The learning rate α starts from 0.1 and was divided by 10 whenever error plateaus, and the models were trained for about 60×10^4 iterations with stochastic gradient descent SGD algorithm. No dropout was used. The input was 224×224 random crop of an image or its horizontal flip from its shorter side, and then normalized by subtracting mean per pixel, and also standard color augmentations were used as in [5]. Figure 3.8 shows training and testing of ResNet-18 and ResNet-34 on ImageNet dataset.

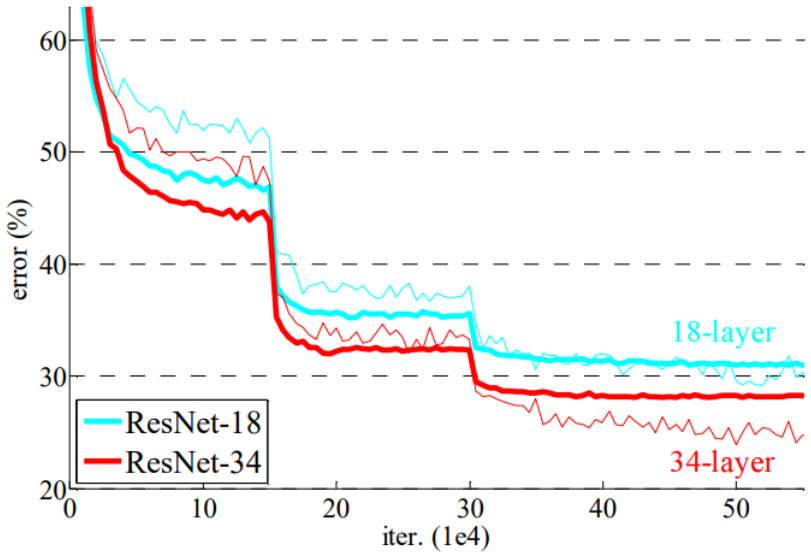


Figure 3.8: Training (light-line) and testing (dark-line) of ResNet-18 (27.88% top-1 error) and ResNet-34 (25.03% top-1 error) on ImageNet dataset. [6]

Both of these fully trained models can be obtained from following links:

- ResNet-18 : `https://www.cntk.ai/Models/ResNet/ResNet_18.model`
- ResNet-34 : `https://www.cntk.ai/Models/ResNet/ResNet_34.model`

Transfer learning

As we discussed earlier in section 2.7, Transfer learning is a technique of taking a pretrained model with the weights and biases and fine tuning it with your own data set. Therefore, we decided to take these two models ResNet-18 and ResNet-32 trained on ImageNet, and finetune it on our dataset of the visual representations of normal and abnormal machine sounds. In order to do that, we took these models and replaced the last fully connected layer of 1000 categories for ImageNet with a fully connected layer of two categories for normal and abnormal sounds. We then retrained it on a series of visual representations of audio in frequency domain. Chapter 4 describes in greater detail about training and testing of our models for anomaly detection.

Chapter 4

Data collection and results analysis

As discussed in previous chapter, main task of the project is to collect audio of a running machine and then convert it into a visual representation in frequency domain and classify it into abnormal and normal categories. This chapter describes the entire process from data collection, to data processing and from training the network to analyzing results.

4.1 Background

We worked in collaboration with Rovema Gmbh. It's an international company that makes packaging equipment. They have several machines for packaging, which use auger dosing method to fill the product in the packages. Following Figure 4.1 shows one of those machines. It consists of a funnel, where product is first mixed in the agitator and then passed down through an auger. Whole system is made out of metal and threads of auger are sharp. On run time, product is first poured on the top of the funnel, where a rotating agitator mixes it and then it is pushed down the funnel.

Figure 4.1: Product mixing and packaging machine at Rovema.

Machine can operate at different speeds between 0-1500 rpm and sometimes due to

increased speed, auger starts to scratch against the wall of vertical tube, and for different products scratching occurs at different rotating speed. Therefore, it hard to find maximum rotating speed limit using traditional methods. Normally, they have workers who after working several years with these machines can tell when scratching occurs but, for a new worker it is very hard to distinguish between scratching and non-scratching sounds because they are very subtle. So practically, they have two problems.

1. How to find the safe limit for rotating speed (rpm) of auger?
2. How to automate this task by using a machine learning technique?

Follwing Figure 4.2 show the internal structure of packaging machine and you can see scratching occurs at the bottom of the funnel.

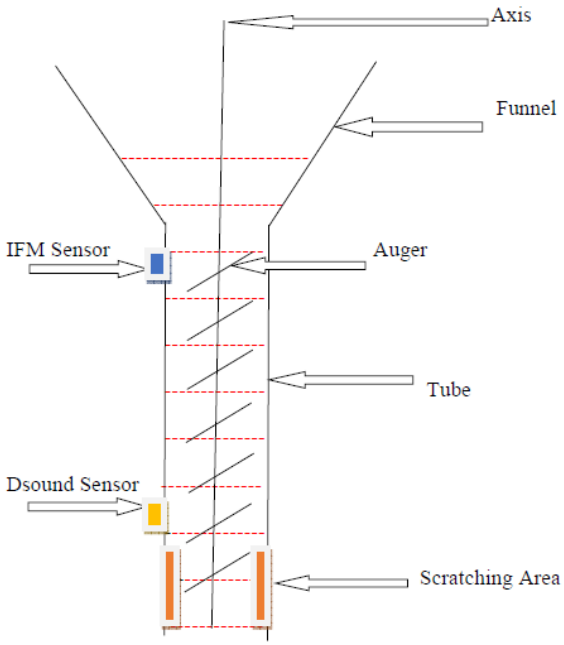


Figure 4.2: Internal structure of product mixing and packaging machine at Rovema. [24]

4.2 Data collection

In order to understand the behavior of machine we used multiple sensors and data obtained from these sensors was stored in a single .dat file with four channels. Figure 4.3 shows the complete architecture of data acquisition system. Channel 1 and 2 store the information of speed and torque respectively. GUI that comes with the machine lets user set different values of speed and torque on run time. Two additional sensors were attached with the tube of auger to get the vibration and sound information. Channel 3 and 4 store information of sound and vibration respectively.

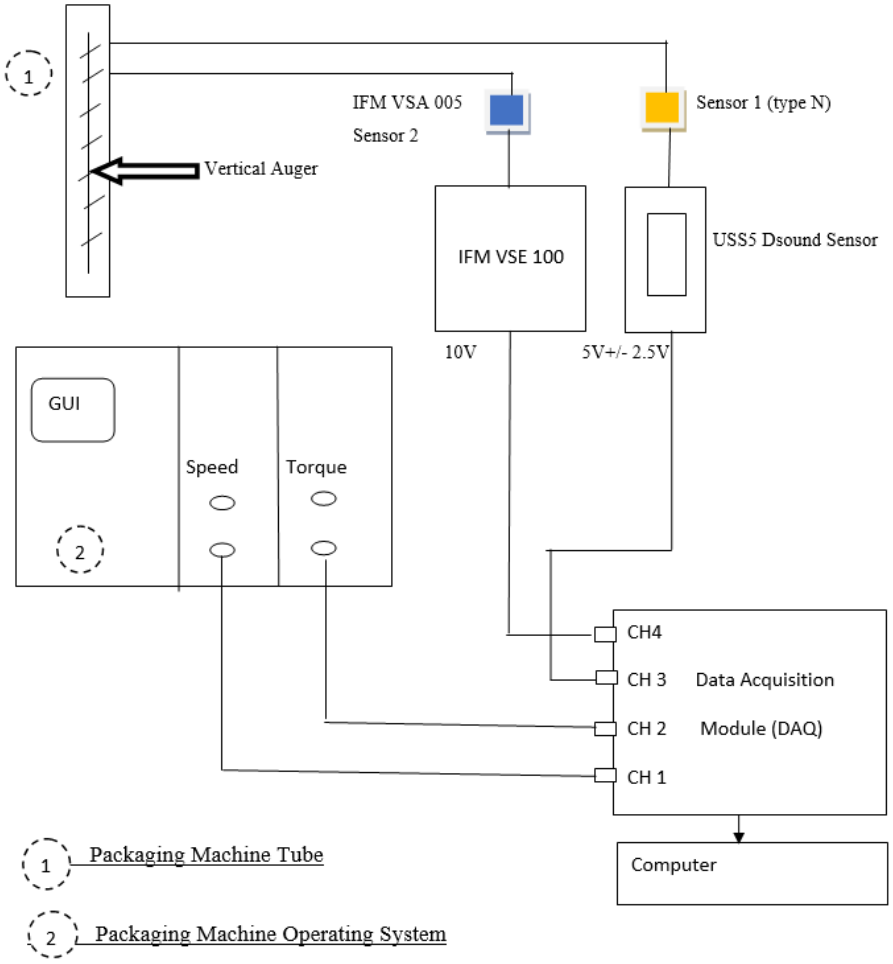


Figure 4.3: Data acquisition system with four channels for speed, torque, sound and vibration. [24]

In our case, we performed ramp measurements with two products Coscus and Powder. We used two augers one new and one old to compare the difference in behavior. The machine was operated between 0 to 1500 rpm by gradually increasing the speed with constant acceleration, and data coming from all four channels was stored in a single .dat file. For two products and two auger we obtained four files in total and all these files were saved under following abbreviations [24]:

- CNA: New auger with Coscus product
- CSA: Old auger with Coscus product
- PNA: New auger with Powder product

- PSA: Old auger with Powder product

Following Figure 4.4 and Figure 4.5 represent data obtained with new and old auger for Coscus product. If you look sound signal more closely, you can see old auger starts scratching at early stage as compared to new auger. Which can also be observed in the spectrogram of these sound signals. Similar data was also obtained for Powder product.

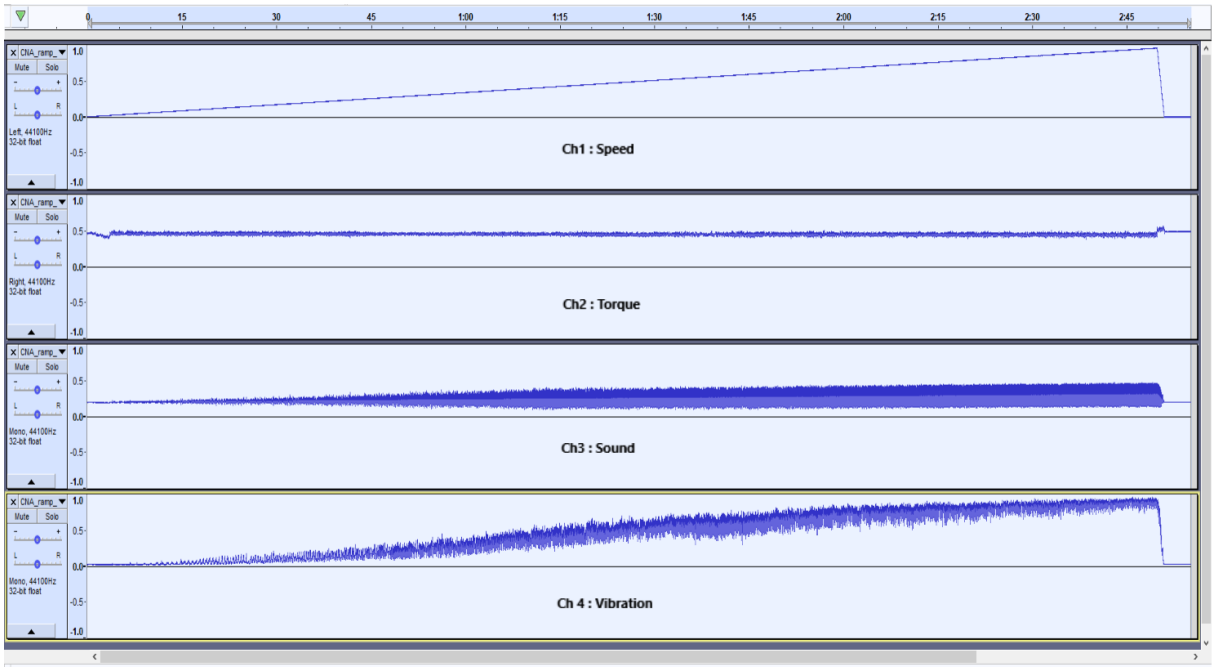


Figure 4.4: CNA: New auger with Coscus product between 0 to 1500 rpm.

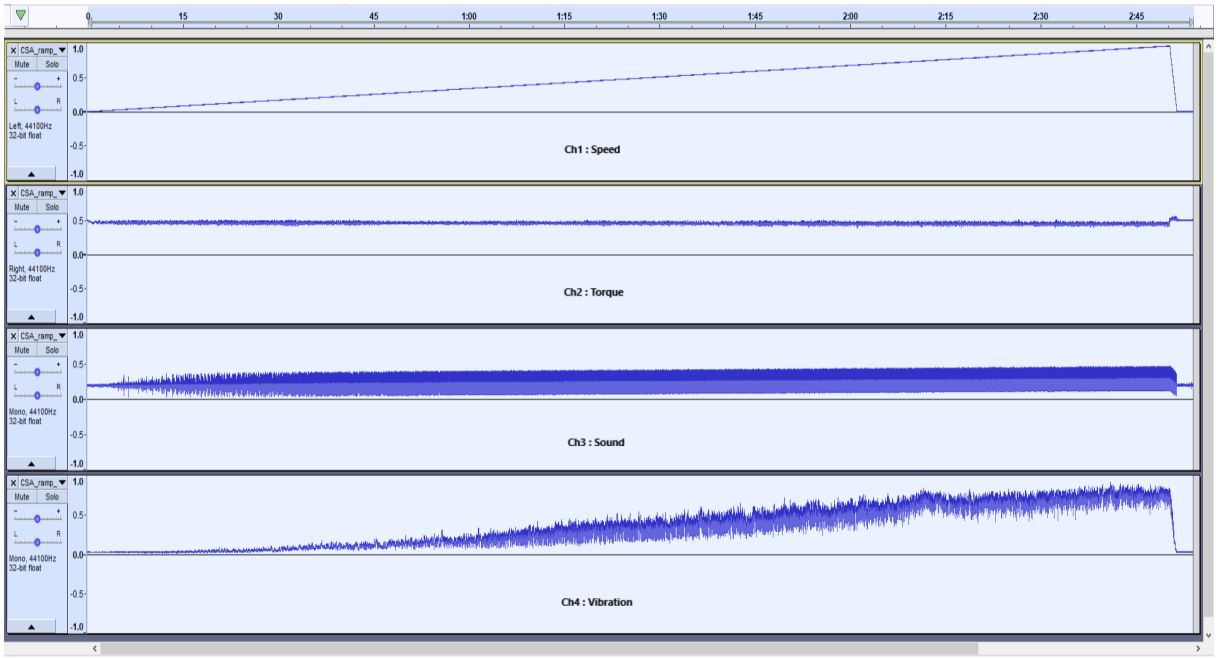


Figure 4.5: CSA: Old auger with Coscus product between 0 to 1500 rpm.

4.3 Audio sampling and frequency domain analysis

All the data samples were collected and stored in .dat file format with four channels as discussed in previous section. We were only interested in data coming from sound sensor, so we exported it in .wav file format. WAV file format is a loss-less audio format standard, which was originally developed by Microsoft and IBM in 1991 for storing audio bitstreams on personal computers. Commonly, it uses pulse code modulation PCM technique to convert analog signal into bits. In PCM, amplitude of an analog signal is sampled at regular instance and each value is adjusted to the nearest value in the range of digital steps. Following Figure 4.6 shows encoding of an analog signal using PCM.

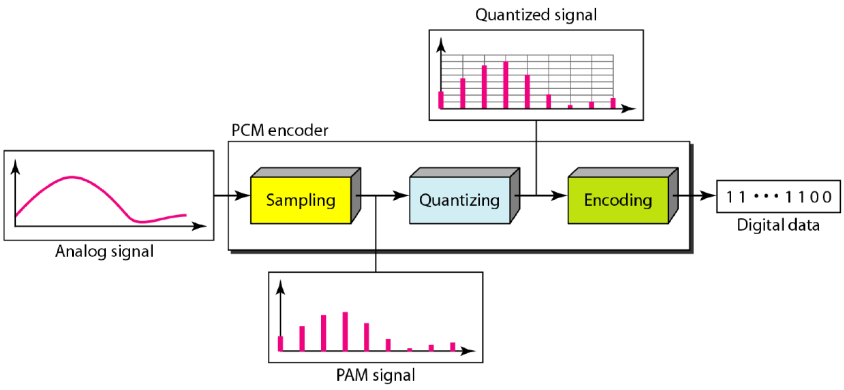


Figure 4.6: Encoding analog audio signal into bit streams using PCM. [25]

In a PCM encoder, every audio signal goes through three steps of sampling, quantizing and encoding. Sampling is a process of recording an analog signal regularly at discrete moments in time. We recorded our audio samples at a rate of 44,100 samples per second. According to the **Nyquist theorem**, sampling rate should be at least twice as high as the highest frequency component in audio signal under observation.

$$f_s \geq 2f_{max} \tag{4.1}$$

So, a 44.1k sampling rate can record the highest frequency of 22.05kHz in an audio signal. After sampling, all recorded values are quantized. Quantizing is a process of adjusting the recorded values to the most nearest step value in time. For a B bit PCM, it has 2^B steps in total. Most commonly 8,16,24 and 32 bit are used. And, the last step is encoding. Encoding is way of converting real values into bits (0,1) so that processor can understand them.

Apart from time domain, it's often helpful to observe signal in frequency domain. In simple words, time domain only tells us how signal changes over time but frequency domain tells us how much of a signal lies within each frequency band over a range of frequencies. Special techniques have been built to convert a time domain signal into its frequency domain. One of the most famous technique is called fourier transform FT. Since we are talking about digital signals, and technique used to convert a digital signal into its frequency domain representation is called discrete fourier transform DFT. There is a fast and efficient algorithm to perform DFT and it is called fast fourier transform FFT. It was first presented by Cooley and Tukey in a paper in 1965 and since then it became a fundamental pillar of digital signal processing [26].

Discrete fourier transform produces its results as individual components in frequency domain which are often called bins. For example, if we take a square wave in time domain,

then it looks like a ramp in frequency domain and DFT looks like series of individual delta functions forming a ramp. As FFT is an optimization algorithm therefore, it comes with its limitations. When you perform FFT of signal, algorithm generally takes a finite set of data and assumes it as one period of a continuous signal. In FFT two end points of waveform in time domain are interpreted as if they are connected together since both time and frequency domains are circular topologies. So when a measured signal is periodic with some integer number of periods, then FFT turns out fine since it matches the assumption. But there may be times when measured signal does not have integer number of periods or it has discontinuities at ends. Theses discontinuities introduce high frequency components which were not present in original signal. These frequencies can be even higher than Nyquist frequncy and are present between zero and half of signals sampling rate. This phenomena is often known as **"spectral leakage"** [27].

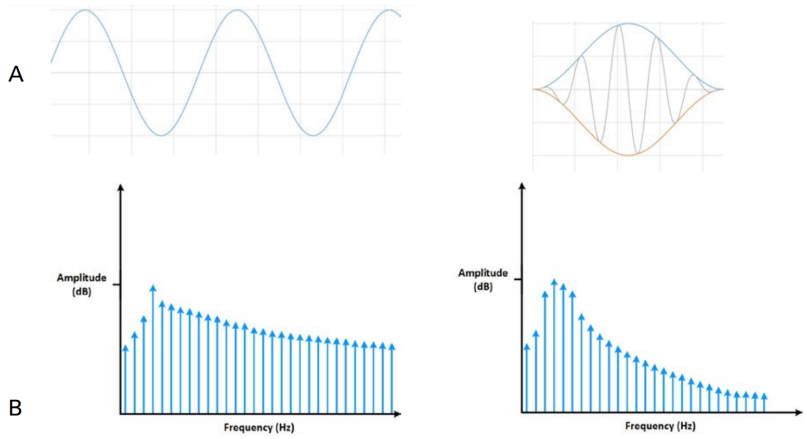


Figure 4.7: Left, discontinued signal without windowing and right, with windowing. [27]

In order to minimize the effect of spectral leakage, a technique called windowing is used. Windowing reduces the amplitude of discontinuities and helps in minimizing its effect in frequency domain. It is simply a multiplication of time series of recorded signal with a window of finite length whose amplitude varies gradually and smoothly towards zero around the edges. There are many types of windowing functions which can applied to a signal. Two of the most famous are Hanning and Hamming window functions. In general Hanning window works well for almost 95% of time. Therefore, we decided to choose Hanning window for our project as well.

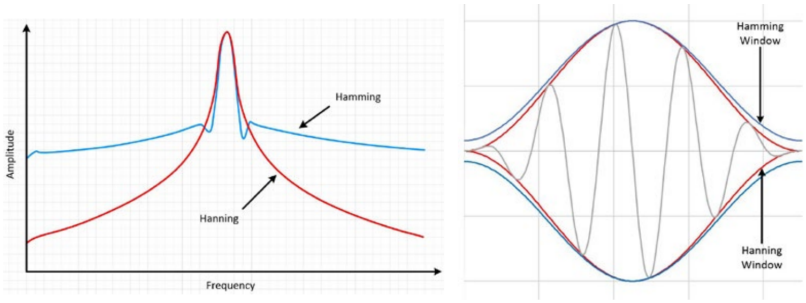


Figure 4.8: Left Hann and Hamm window functions, Right both windows applied to a signal. [27]

After doing time and frequency domain analysis, we observed that scratching sound really adds extra frequency components, which were not present at low "rpm" auger sound signal. Following figures show the frequency domain representation of both new and old

auger for two products “Coscus” and “Powder”.

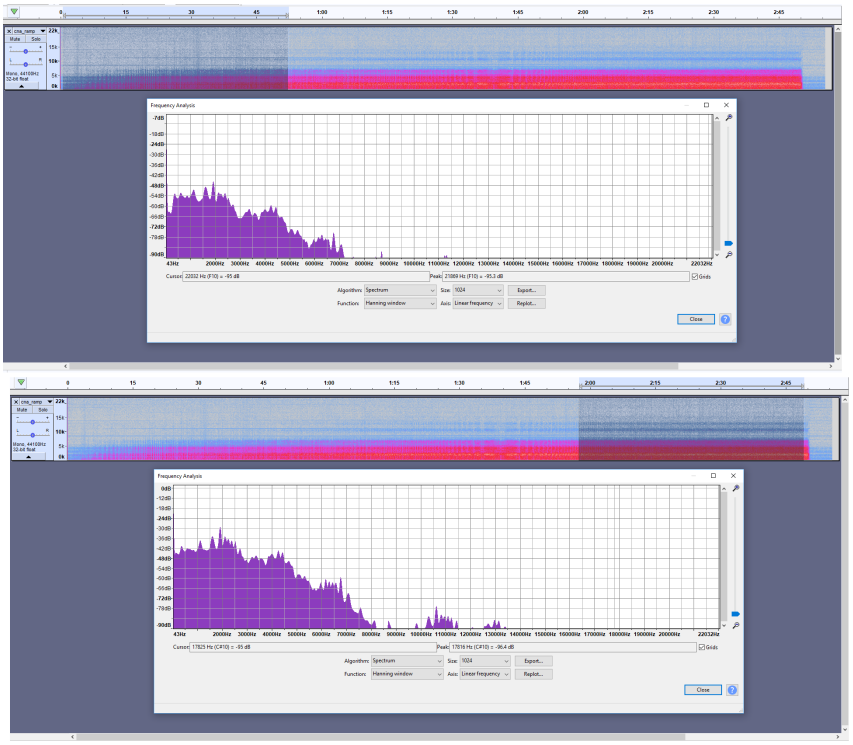


Figure 4.9: CNA - New auger with Coscus, Top at low rpm bottom at high rpm.

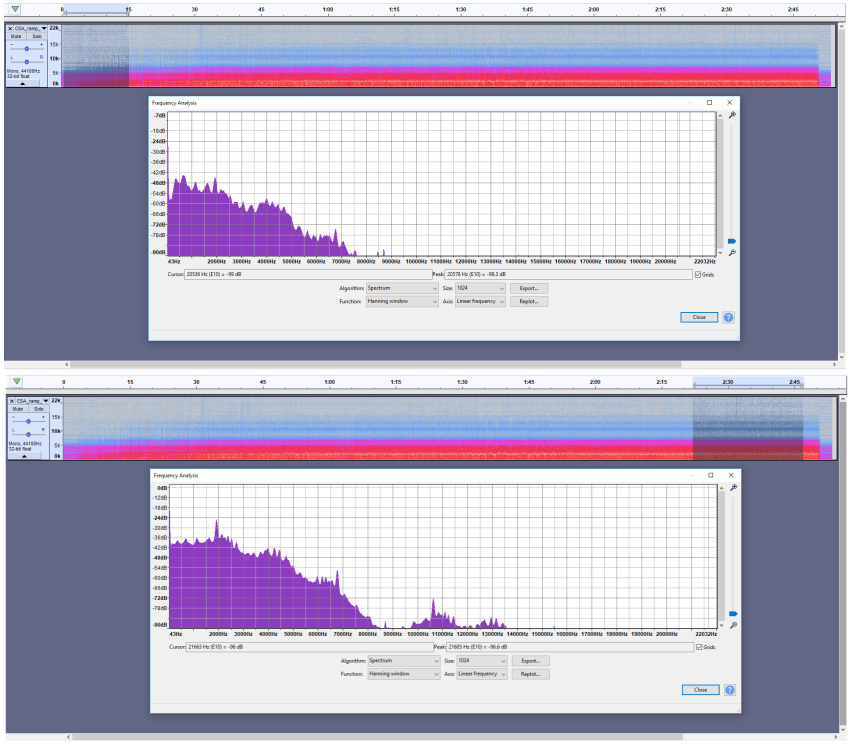


Figure 4.10: CSA - Old auger with Coscus, Top at low rpm and bottom at high rpm.

As you can see both new auger Figure 4.9 and old auger Figure 4.10 for product **Coscus** show new frequencies between (9.5k - 13.5k Hz) at high rpm, the only difference is old

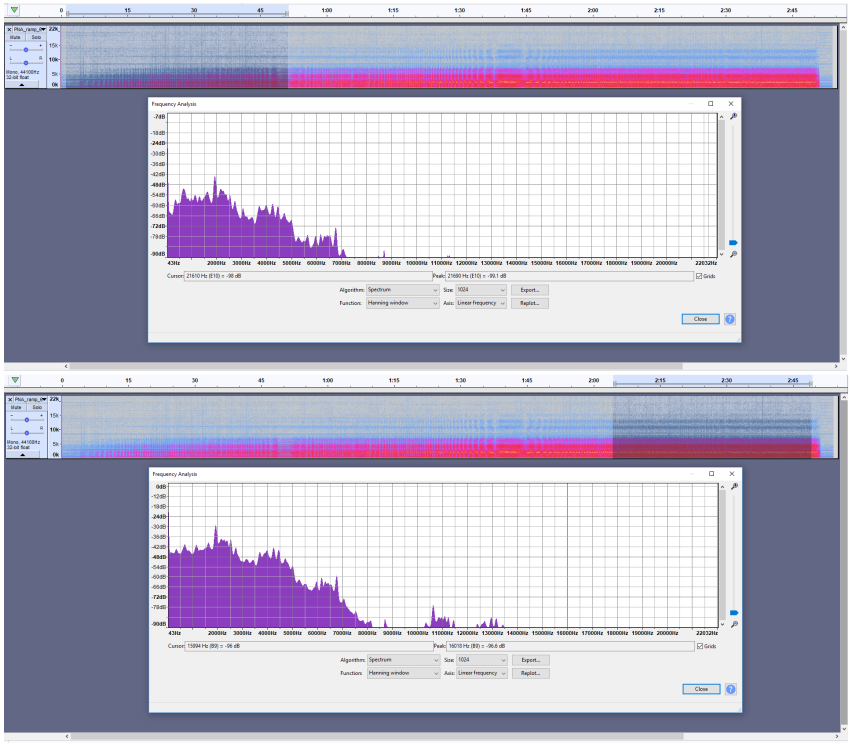


Figure 4.11: PNA - New auger with Powder, Top at low rpm bottom at high rpm.

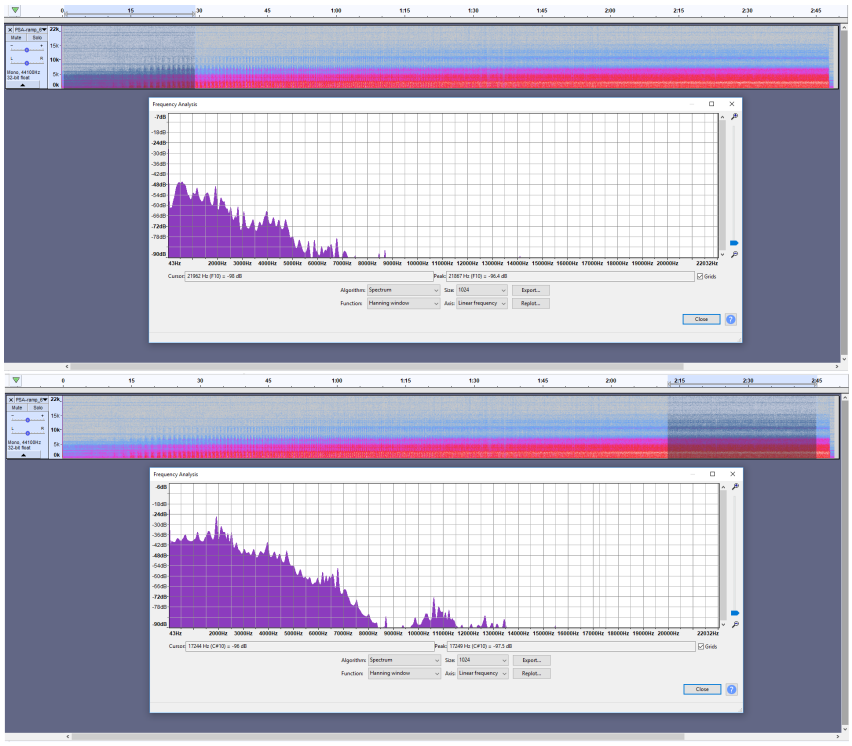


Figure 4.12: PSA - Old auger with Powder, Top at low rpm bottom at high rpm.

auger starts scratching earlier whereas new auger scratches at considerably higher rate of "rpm". Similar trend is observed in case of Powder product for both new Figure 4.11 and old Figure 4.12 auger.

4.4 Visual representations of audio

As we saw in previous section, both scratching and non scratching sounds have different signature. Hence both can be represented visually and we can perform image recognition to distinguish a normal sound from an abnormal sound. The technique of visual representation of sound is called **spectrogram**. It is 2D representation of sound with time on one axis and frequency on the other axis, and where pixel value represents the intensity value of frequencies. The principal behind is to compute successive DFTs and plot them against time with relative value of each bin of DFT depicted on a color scale.

The frequency and temporal resolutions of a spectrogram are linked in inverse way. For example, increasing the size of window will increase the frequency resolution but decreases the resolution in time, and similarly decreasing the window size increases the temporal resolution but at the expense of frequency resolution. However, we can counteract this two dimensional precision limit problem with overlapping DFT windows. This overlap is usually set up in percentage. An overlap of 50 percent will increase the time resolution by a factor of 2 while keeping the frequency resolution same. The only down side it has is, that it will double the number of DFTs and hence more computations. So it is often advised to keep the overlap at a reasonable value [28]. Each spectrogram has three values as following:

- time : all the values of time axis
- frequency: all the values of frequency axis
- amp: all values of successive FFTs decompositions

In our project, we took (4 seconds) segment of the audio and walked through that segment by performing FFT on it with hamming window of size (2048) and overlap of 50%. We repeated same process over total length of the audio by sliding (4 second) window to the right by 1 second. We took all these matrix (time, frequency) values and normalized them to (0 - 1000). And then we plotted them using grey scale with frequency resolution on the y axis and time resolution on x axis. Then we converted them into (256×256) images and saved them in .png format. Following images are spectrogam of (4 second) chunks of two normal and two anomalous (scratching) sounds. And you can clearly see the visual difference in both.

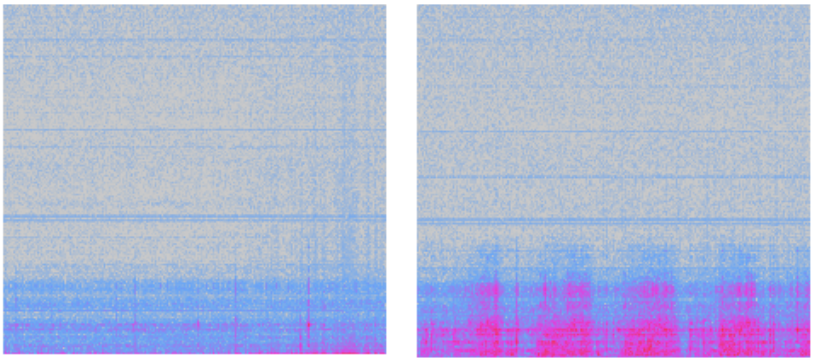


Figure 4.13: Spectrogram of normal sound of auger with Coscus

As you can see in Figure 4.14, it has some high frequencies in it, while in case of Figure 4.13 only low frequency components are present in the signal. In order to perform all these above mentioned operations we used following python libraries:

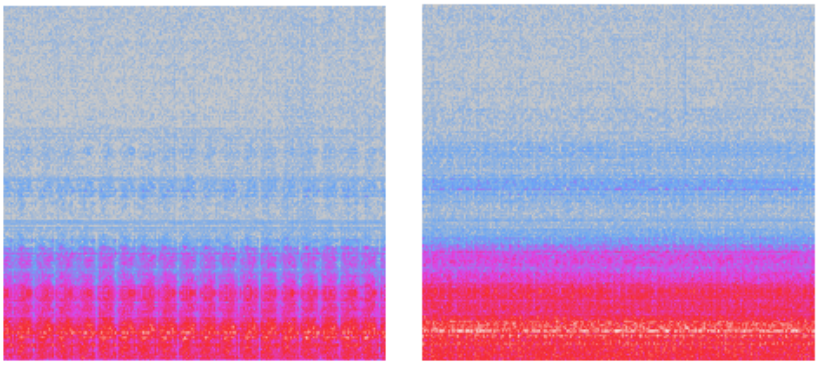


Figure 4.14: Spectrogram of abnormal (scratching) sound of auger with Coscus

1. **numpy**
   - *numpy.io.wave()* - to read wav file, it outputs sampling rate and data.
   - *numpy.hanning()* - to use hanning window funtion to perform FFT.
   - *numpy.fft.rfft()* - to perform FFT and since we are only interested in magnitude and all the frequencies of fft are skew symmetric, therefore we used rfft()
2. **scikit**
   - *sklearn.preprocessing.MinMaxScaler()* - to scale data within specified range.
3. **scipy**
   - *scipy.misc.imsave()* - to save images in specified format.
   - *scipy.misc.imresize()* - to set the image according to required dimensions.
4. **matplolib**
   - *matplotlib.cm()* - to choose color map scheme, gist_ncr in our case.

4.4.1 Map files

All the images were stored in local .\images directory and were named simply as $(img0, img1 \ldots imgN)$ for seconds range $(0, \ldots, N)$. As we have decided to use microsoft's library called CNTK for our project and the model created by it takes inputs in form of map files. A map file simply tells our model the location and label of corresponding image. Reason we need to create map files is that during training time model does not take images one by one. Instead a series of images are fed at once in form of a minibatch.

We divided our complete dataset into two sets called training set and testing set. Training set consisted of 70% and testing set consisted of 30% of our total data. Therefore, we created two map files one for training set and one for test set. We named them as **train_map.txt** and **test_map.txt** and stored them in the same .\images directory. Noraml sound images were labeled as 0, whereas anomalous sound images were labeled as 1. Location and labels were all tab delimited. Following Figure 4.15 is an extract from a map file and you can see how the address and labels are stored correspondingly.

```
D:\Master-Thesis\Juypyter-Notebooks\Thesis_Project\images/img34.png     0
D:\Master-Thesis\Juypyter-Notebooks\Thesis_Project\images/img35.png     0
D:\Master-Thesis\Juypyter-Notebooks\Thesis_Project\images/img37.png     1
D:\Master-Thesis\Juypyter-Notebooks\Thesis_Project\images/img38.png     1
D:\Master-Thesis\Juypyter-Notebooks\Thesis_Project\images/img40.png     1
```

Figure 4.15: Map file with exact location and corresponding label of the image.

4.5 Deep learning library

As discussed earlier in chapter 3, we took two pretrained models ResNet-18 and ResNet-34 trained on ImagNet. Now we want to reuse them at our problem of anomaly detection. We forced these models to learn on our new dataset by training on the new images we created from our audio samples. In order to retrain these models, we decided to use Microsoft Cognitive Toolkit library called CNTK.

4.5.1 CNTK

Cognitive toolkit is an open-source framework initially developed by Microsoft for executing, training and describing computational networks. It is also a framework for describing deep neural networks and it has a built-in support for both CPUs and GPUs. The most recent version for python is 2.3. For setting it up on your machine please refer to this link: Setup CNTK on your machine.

Apart from CNTK, there are some other famous frameworks too e.g. TensorFlow, Caffe and Theano. The reason we decided to choose CNTK are as following [29]:

- Speed: CNTK is much faster than TensorFlow and in some cases even 5-10x faster. Research paper by Shaohuai Shi called "Benchmarking State-of-the-Art Deep Learning Software Tools" suggests that CNTK is usually 2-3 times faster than TensorFlow for image-related tasks.

- Accuracy: CNTK was designed by paying lot of attention for tracking down bugs and to achieve state of the art accuracy. For example, automatic batching algorithm allows us to pack sequences with different length and allows better randomization, which improves accuracy by 1-2% as compared to naive data packing.

- API design: Python API of CNTK contains implementations of both low level and high level. High level API design is very compact and intuitive with actual implementation and easy to follow even for beginners.

- Scalability: Being able to scale across multiple GPUs is very important in deep learning models where tasks can face millions of training examples. And you can easily change from single GPU to multiple GPUs by merely changing few lines of code.

- Built-in readers: In deep learning, we have lot of training data and one the problems mostly faced is that it does not fit in the RAM or on a single machine. Or if it fits in the RAM then amateur training loop spends too much time sending data from RAM to GPU. CNTK has built in readers which typically address all these problems.

4.5.2 Basic modules of CNTK

CNTK performs operations in a form of computation graph just like other famous libraries e.g. Theano or TensorFlow. A computation graph consists of nodes and edges.

A node is simply a variable which holds some value and an edge is an operation performed on that variable. In case of deep learning, variables in computation graph are often called tensors. A tensor is simply a high dimensional array e.g. rank 0 tensor is simply a scalar, rank 1 tensor is a vector and rank 2 tensor is a matrix. Tensor can be even of higher order. For example, a batch of 16 images can be thought as a tensor of order $(16 \times 3 \times 256 \times 256)$. This can be easily visualized as computational graph. Let's consider following graph of computations. For example, variable a directly effects variable c and the amount of effect can be represented in from of partial derivative.

Back propagation is a core algorithm in deep learning and it involves computing derivatives in reverse order to find global minimum. Consider following figure, if we want to understand how variable a and b affects variable e we have to calculate partial derivatives all the way up as shown in Figure 4.16.

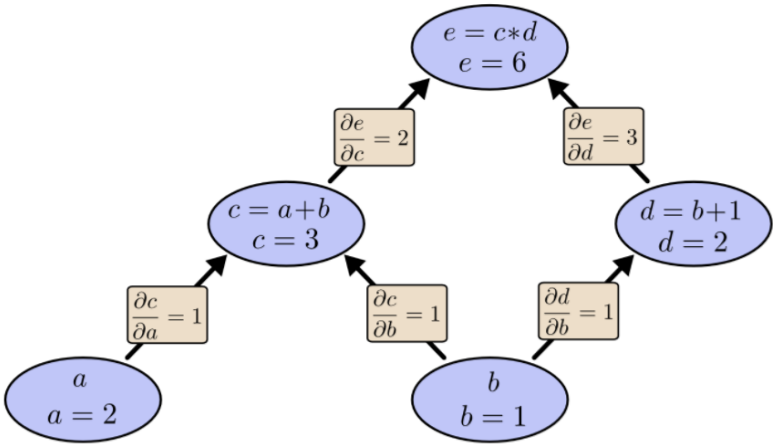


Figure 4.16: Computational graph for forward propagation. [30]

And now if we want to calculate how change in e effects all the nodes in the graph, we have to take partial derivatives in reverse mode as shown in following Figure 4.17.

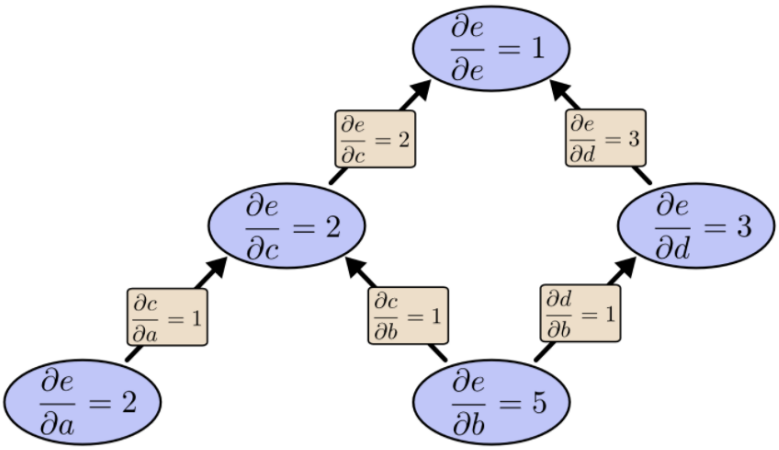


Figure 4.17: Computational graph for backward propagation. [30]

CNTK offers many built-in modules and in our project, we used following modules of the library [31].

1. **Cntk.io**: This module handles all input and output operations. We used following methods of this module.
    - *MinibatchSource*: It creates minibatches for specified parameters, which is later fed to the model for training.
    - *ImageDeserializer*: It configures the image reader which is responsible for reading images from map files with locations and corresponding labels.
    - *StreamDefs*: It configures the stream for the use with ImageDeserializer.
2. **Cntk.layers**: Layers are building blocks of any model and this module helps to configure them. We used following methods of this module.
    - *Dense*: It is a factory method which helps to construct a fully connected layer with activation function, weights and biases.
3. **Cntk.learner**: This is one of the most important package of CNTK which sets training parameters.
    - *momentum_sgd*: It creates instance for momentum SGD algorithm.
    - *learning_rate_schedule*: It schedules the learning rate for training. We can set different values of learning rate for different minibatches and epochs.
    - *momentum_schedule*: It schedules the momentum value during training process.
4. **Cntk.losses**: This module provides helper functions to keep track of loss and cross entropy values during training.
    - *cross_entropy_with_softmax*: This calculates cross entropy between softmax of output vector and target vector.
5. **Cntk.metrics**: This module provides helper functions to calculate classification error.
    - *classification_error*: This is used during evaluation and computes classification error.
6. **Cntk.logging**: This is built-in modules which helps to perform logging operations.
    - *ProgressPrinter*: This is a class which logs all the loss and cross entropy values.
    - *Graph*: This is sub-module which provides helper function for creating graph of training progress.

4.6 Training and analyzing results

After preparing all the images of a selected audio as discussed in previous section, we are now ready to train our model. We performed our experiment with two models ResNet-18 and ResNet-34. We downloaded and saved both models in .\models directory. Similarly, all the audio files were stored in .\audio and all the corresponding images of a particular audio file were stored in .\images directory.

Both of these models were already trained on ImageNet, so they had 1000 outputs in the output layer. In our case we had only two outputs "Normal" and "Scratching" sound, so we searched the last output layer and replaced it with two outputs with corresponding labels. We labeled the normal and scratching section of all the audio files very carefully by looking into frequency domain as following.

- **cna_ramp.wav** : Total length $0 - 165s$ and scratching $65 - 165s$
- **pna_ramp.wav** : Total length $0 - 165s$ and scratching $75 - 165s$
- **csa_ramp.wav** : Total length $0 - 165s$ and scratching $25 - 165s$
- **psa_ramp.wav** : Total length $0 - 165s$ and scratching $55 - 165s$

All the stored images also had two labels 0 and 1 for normal and scratching sound respectively, and they were saved in the map file as discussed in section 4.4.1. In order to train the models, we used a GPU GeForce GTX 950M and NVIDIA GPU computing toolkit CUDA v.8.0. Detailed build info is presented in following Figure 4.18.

```
Selected GPU[0] GeForce GTX 950M as the process wide default device.
----------------------------------------------------------------------
Build info:

                Built time: Jul 31 2017 03:29:10
                Last modified date: Wed Jul 26 04:19:54 2017
                Build type: Release
                Build target: GPU
                With 1bit-SGD: no
                With ASGD: yes
                Math lib: mkl
                CUDA_PATH: C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0
                CUB_PATH: c:\local\cub-1.4.1
                CUDNN_PATH: C:\local\cudnn-8.0-windows10-x64-v6.0.21\cuda
                Build Branch: HEAD
                Build SHA1: 5643a5619097b125f49e629f5bdbbc5a6ceedcf0
                Built by svcphil on DPHAIM-25
                Build Path: C:\jenkins\workspace\CNTK-Build-Windows\Source\CNTKv2LibraryDll\
                MPI distribution: Microsoft MPI
                MPI version: 7.0.12437.6
----------------------------------------------------------------------
[I 15:17:11.308 NotebookApp] Saving file at /Thesis_Project/Audio_Anomaly_Detector.ipynb
[I 15:17:16.753 NotebookApp] 302 GET /notebooks/audio/cna_ramp.wav (::1) 2.01ms
----------------------------------------------------------------------
```

Figure 4.18: Build info of device for training models.

Since we had two augers (new and old) in our experiment, so we decided to train separate models for both. For training we used cna_ramp.wav for new and csa_ramp.wav for old auger. We split 165 images into the 2/3 (total 110) for training and 1/3 (total 55) for testing. We saved our models under following names:

1. **18 layer**
    - **auger_18_new** : 18 layer model for new auger trained on cna_ramp.wav
    - **auger_18_old** : 18 layer model for old auger trained on csa_ramp.wav

2. **34 layer**

   - **auger_34_new** : 34 layer model for new auger trained on cna_ramp.wav
   - **auger_34_old** : 34 layer model for old auger trained on csa_ramp.wav

We went further and tested the performance of our models for audio of powder products pna_ramp.wav and psa_ramp.wav, and documented the results.

4.6.1 Experiment with 18 layer model ResNet-18

We trained and tested 18 layer ResNet model for both old and new auger with cna_ramp.wav and csa_ramp.wav respectively. For training 18 layer model we used following parameters.

- Total number of epochs = 10
- Minibatch size = 10
- Learning rate = 0.01
- Momentum = 0.9
- L2 Regularization = 0.0005

The size of training set was 110 images and size of the test set was 55. Input dimension of the model was $(3 \times 224 \times 224)$ and output was 2 classes with softmax probabilities for normal and scratching.

18-layer model for "new auger"

auger_18_new is the 18 layer model for new auger trained on cna_ramp.wav. We trained this model with the parameters as described above. Following Table 4.1 and Figure 4.19 show the details of training process. After training the model we tested it with our test set of 55 images from cna_ramp.wav and obtained results as shown in Figure 4.20. We further tested its performance with 55 random images of audio from powder product pna_ramp.wav and obtained results as shown in Figure 4.21. So in Figure 4.20 you can see it predicted 52 images correctly and labeled 2 images wrongly and for powder product you can see in the Figure 4.21 it predicted 49 images correctly and 6 images wrongly.

```
Training transfer learning model for 10 epochs (epochSize = 110).
Training 15898178 parameters in 68 parameter tensors.
Finished Epoch[1 of 10]: [Training] loss = 0.675340 * 110, metric = 25.45% * 110 7.290s ( 15.1 samples/s);
Finished Epoch[2 of 10]: [Training] loss = 0.103662 * 110, metric = 4.55% * 110 2.084s ( 52.8 samples/s);
Finished Epoch[3 of 10]: [Training] loss = 0.024449 * 110, metric = 0.91% * 110 2.072s ( 53.1 samples/s);
Finished Epoch[4 of 10]: [Training] loss = 0.069667 * 110, metric = 1.82% * 110 2.062s ( 53.4 samples/s);
Finished Epoch[5 of 10]: [Training] loss = 0.012901 * 110, metric = 0.00% * 110 2.085s ( 52.8 samples/s);
Finished Epoch[6 of 10]: [Training] loss = 0.013070 * 110, metric = 0.00% * 110 2.074s ( 53.0 samples/s);
Finished Epoch[7 of 10]: [Training] loss = 0.002669 * 110, metric = 0.00% * 110 2.078s ( 52.9 samples/s);
Finished Epoch[8 of 10]: [Training] loss = 0.001932 * 110, metric = 0.00% * 110 2.097s ( 52.5 samples/s);
Finished Epoch[9 of 10]: [Training] loss = 0.001962 * 110, metric = 0.00% * 110 2.095s ( 52.5 samples/s);
Finished Epoch[10 of 10]: [Training] loss = 0.044288 * 110, metric = 2.73% * 110 2.064s ( 53.3 samples/s);
```

Table 4.1: Training process of auger_18_new model for 10 epochs.

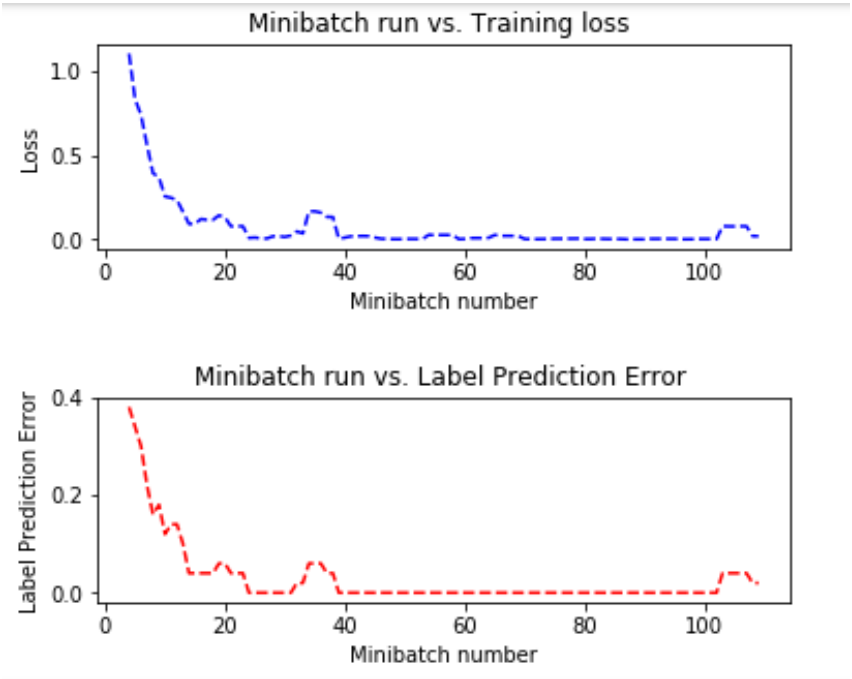


Figure 4.19: Training graph of auger_18_new model.

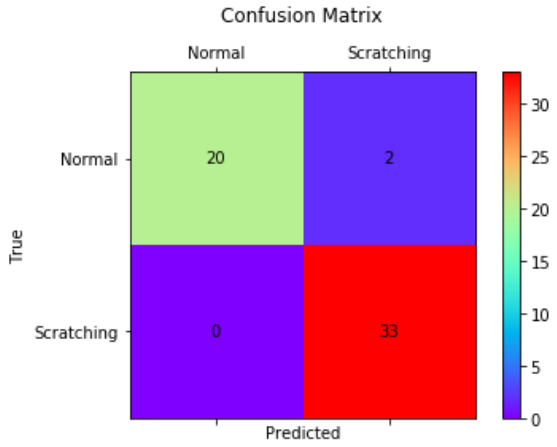


Figure 4.20: Testing, 53/55 test images predicted correctly from **coscus** cna_ramp.wav.

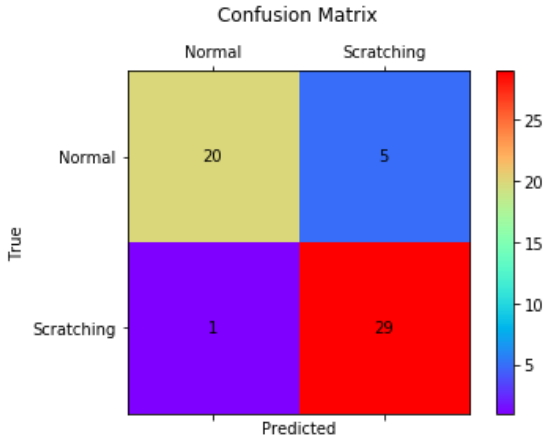


Figure 4.21: Testing, 49/55 test images predicted correctly from **powder** pna_ramp.wav.

18-layer model for "old auger"

auger_18_old is the 18 layer model for old auger trained on csa_ramp.wav. We trained this model with the parameters as described before. Following Table 4.2 and Figure 4.22 show the details of training process. After training the model we tested it with our test set of 55 images from csa_ramp.wav and obtained results as shown in Figure 4.23. We further tested its performance with 55 random images of audio from powder product psa_ramp.wav and obtained results as shown in Figure 4.24. So in Figure 4.23 you can see it predicted 51 images correctly and labeled 4 images wrongly and for powder product you can see in the Figure 4.24 it predicted 47 images correctly and 8 images wrongly.

```
Training transfer learning model for 10 epochs (epochSize = 110).
Training 15898178 parameters in 68 parameter tensors.
Finished Epoch[1 of 10]: [Training] loss = 0.767972 * 110, metric = 34.55% * 110 7.319s ( 15.0 samples/s);
Finished Epoch[2 of 10]: [Training] loss = 0.042112 * 110, metric = 0.91% * 110 2.075s ( 53.0 samples/s);
Finished Epoch[3 of 10]: [Training] loss = 0.009754 * 110, metric = 0.00% * 110 2.049s ( 53.7 samples/s);
Finished Epoch[4 of 10]: [Training] loss = 0.010068 * 110, metric = 0.00% * 110 2.077s ( 53.0 samples/s);
Finished Epoch[5 of 10]: [Training] loss = 0.003641 * 110, metric = 0.00% * 110 2.084s ( 52.8 samples/s);
Finished Epoch[6 of 10]: [Training] loss = 0.005322 * 110, metric = 0.00% * 110 2.075s ( 53.0 samples/s);
Finished Epoch[7 of 10]: [Training] loss = 0.002149 * 110, metric = 0.00% * 110 2.083s ( 52.8 samples/s);
Finished Epoch[8 of 10]: [Training] loss = 0.003658 * 110, metric = 0.00% * 110 2.074s ( 53.0 samples/s);
Finished Epoch[9 of 10]: [Training] loss = 0.002528 * 110, metric = 0.00% * 110 2.081s ( 52.9 samples/s);
Finished Epoch[10 of 10]: [Training] loss = 0.001378 * 110, metric = 0.00% * 110 2.080s ( 52.9 samples/s);
```

Table 4.2: Training process of auger_18_old model for 10 epochs.

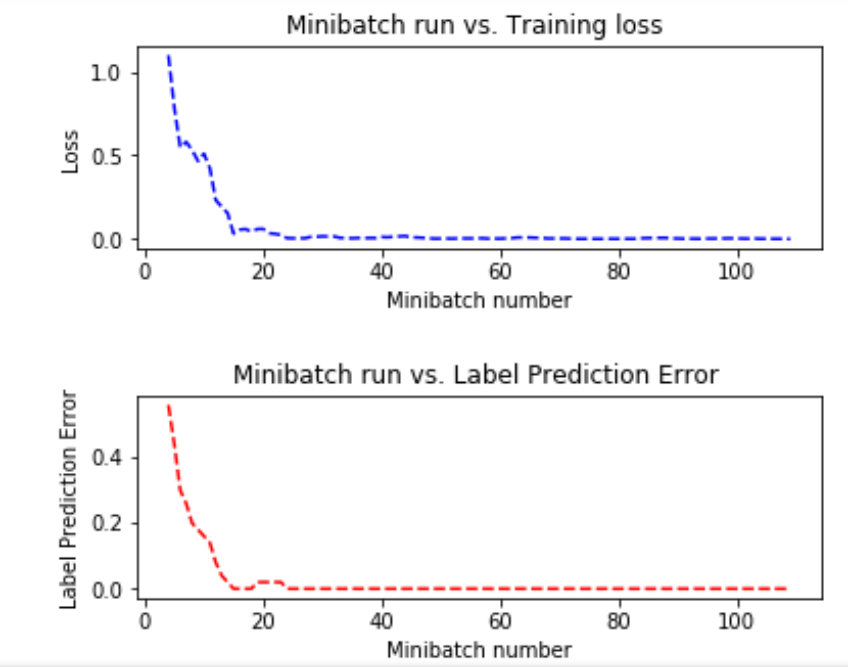


Figure 4.22: Training graph of auger_18_old model.

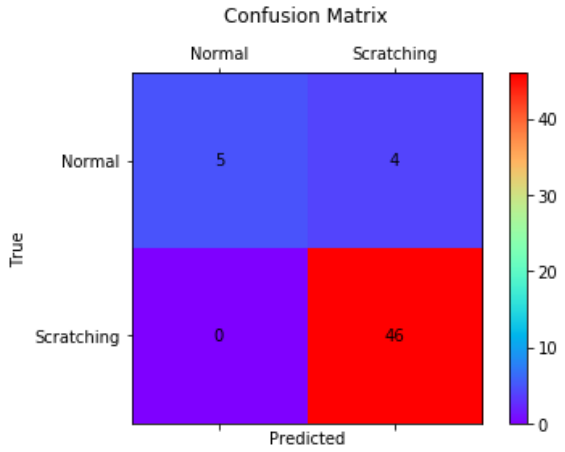


Figure 4.23: Testing, 51/55 test images predicted correctly from **coscus** csa_ramp.wav.

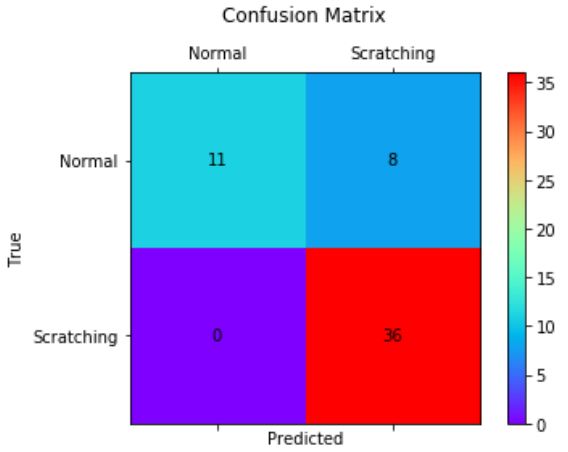


Figure 4.24: Testing, 47/55 test images predicted correctly from **powder** psa_ramp.wav.

4.6.2 Experiment with 34 layer model ResNet-34

We trained and tested 34 layer ResNet model for both old and new auger with cna_ramp.wav and csa_ramp.wav respectively. For training 34 layer model we used following parameters.

- Total number of epochs = 15
- Minibatch size = 10
- Learning rate = 0.01
- Momentum = 0.9
- L2 Regularization = 0.0005

The size of training set was 110 images and size of the test set was 55. Input dimension of the model was $(3 \times 224 \times 224)$ and output was 2 classes with softmax probabilities for normal and scratching.

34-layer model for "new auger"

auger_34_new is the 34 layer model for new auger trained on cna_ramp.wav. We trained this model with the parameters as described above. Following Table 4.3 and Figure 4.25

show the details of training process. After training the model we tested it with our test set of 55 images from cna_ramp.wav and obtained results as shown in Figure 4.26. We further tested its performance with 55 random images of audio from powder product pna_ramp.wav and obtained results as shown in Figure 4.27. So in Figure 4.26 you can see it predicted 54 images correctly and labeled 1 images wrongly and for powder product you can see in the Figure 4.27 it predicted 49 images correctly and 6 images wrongly.

```
Training transfer learning model for 15 epochs (epochSize = 110).
Training 21285698 parameters in 110 parameter tensors.
Finished Epoch[1 of 15]: [Training] loss = 0.835481 * 110, metric = 33.64% * 110 11.090s (  9.9 samples/s);
Finished Epoch[2 of 15]: [Training] loss = 0.163774 * 110, metric = 5.45% * 110 3.167s ( 34.7 samples/s);
Finished Epoch[3 of 15]: [Training] loss = 0.003764 * 110, metric = 0.00% * 110 3.127s ( 35.2 samples/s);
Finished Epoch[4 of 15]: [Training] loss = 0.053829 * 110, metric = 1.82% * 110 3.157s ( 34.8 samples/s);
Finished Epoch[5 of 15]: [Training] loss = 0.047927 * 110, metric = 1.82% * 110 3.139s ( 35.0 samples/s);
Finished Epoch[6 of 15]: [Training] loss = 0.011860 * 110, metric = 0.00% * 110 3.139s ( 35.0 samples/s);
Finished Epoch[7 of 15]: [Training] loss = 0.003214 * 110, metric = 0.00% * 110 3.140s ( 35.0 samples/s);
Finished Epoch[8 of 15]: [Training] loss = 0.002601 * 110, metric = 0.00% * 110 3.129s ( 35.2 samples/s);
Finished Epoch[9 of 15]: [Training] loss = 0.003899 * 110, metric = 0.00% * 110 3.118s ( 35.3 samples/s);
Finished Epoch[10 of 15]: [Training] loss = 0.042980 * 110, metric = 0.91% * 110 3.143s ( 35.0 samples/s);
Finished Epoch[11 of 15]: [Training] loss = 0.002166 * 110, metric = 0.00% * 110 3.139s ( 35.0 samples/s);
Finished Epoch[12 of 15]: [Training] loss = 0.000994 * 110, metric = 0.00% * 110 3.129s ( 35.2 samples/s);
Finished Epoch[13 of 15]: [Training] loss = 0.032731 * 110, metric = 0.91% * 110 3.150s ( 34.9 samples/s);
Finished Epoch[14 of 15]: [Training] loss = 0.001513 * 110, metric = 0.00% * 110 3.141s ( 35.0 samples/s);
Finished Epoch[15 of 15]: [Training] loss = 0.000996 * 110, metric = 0.00% * 110 3.141s ( 35.0 samples/s);
```

Table 4.3: Training process of auger_34_new model for 15 epochs.

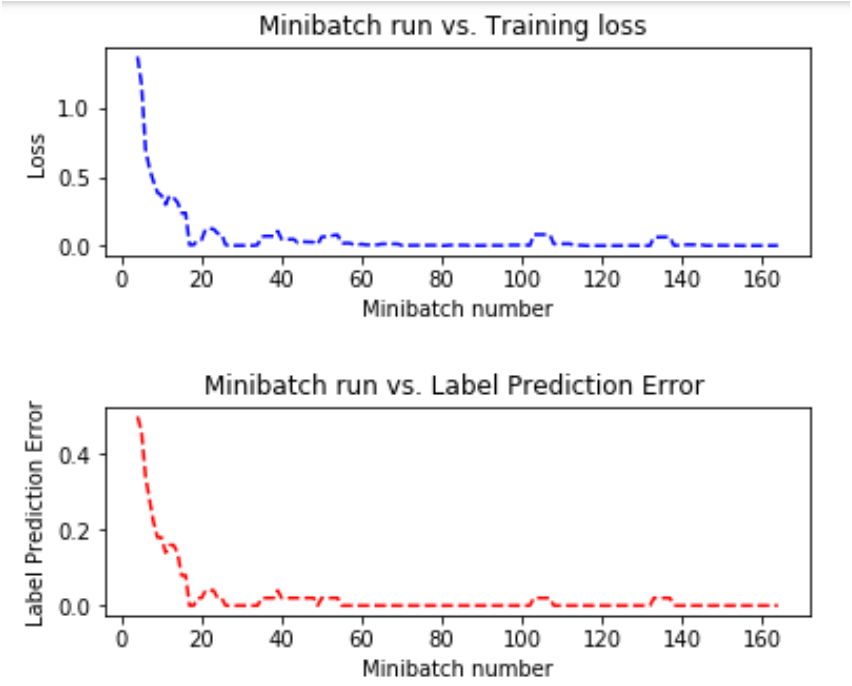


Figure 4.25: Training graph of auger_34_new model.

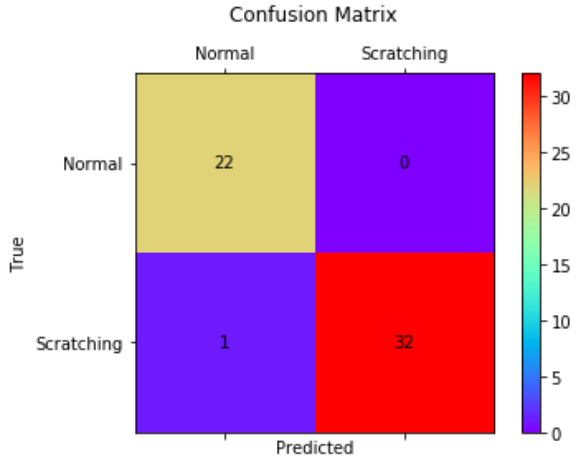


Figure 4.26: Testing, 54/55 test images predicted correctly from **coscus** cna_ramp.wav.

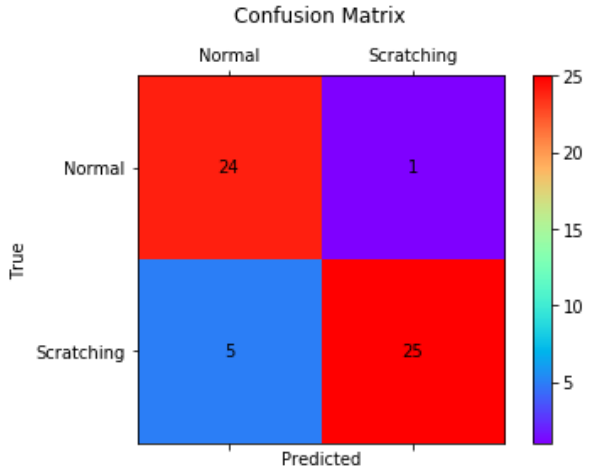


Figure 4.27: Testing, 49/55 test images predicted correctly from **powder** pna_ramp.wav.

34-layer model for "old auger"

auger_34_old is the 34 layer model for old auger trained on csa_ramp.wav. We trained this model with the parameters as described before. Following Table 4.4

```
Training transfer learning model for 15 epochs (epochSize = 110).
Training 21285698 parameters in 110 parameter tensors.
Finished Epoch[1 of 15]: [Training] loss = 0.533680 * 110, metric = 22.73% * 110 11.049s ( 10.0 samples/s);
Finished Epoch[2 of 15]: [Training] loss = 0.090602 * 110, metric = 4.55% * 110 3.130s ( 35.1 samples/s);
Finished Epoch[3 of 15]: [Training] loss = 0.014205 * 110, metric = 0.00% * 110 3.131s ( 35.1 samples/s);
Finished Epoch[4 of 15]: [Training] loss = 0.006953 * 110, metric = 0.00% * 110 3.166s ( 34.7 samples/s);
Finished Epoch[5 of 15]: [Training] loss = 0.003390 * 110, metric = 0.00% * 110 3.151s ( 34.9 samples/s);
Finished Epoch[6 of 15]: [Training] loss = 0.013089 * 110, metric = 0.91% * 110 3.147s ( 35.0 samples/s);
Finished Epoch[7 of 15]: [Training] loss = 0.003988 * 110, metric = 0.00% * 110 3.130s ( 35.1 samples/s);
Finished Epoch[8 of 15]: [Training] loss = 0.004020 * 110, metric = 0.00% * 110 3.123s ( 35.2 samples/s);
Finished Epoch[9 of 15]: [Training] loss = 0.003755 * 110, metric = 0.00% * 110 3.163s ( 34.8 samples/s);
Finished Epoch[10 of 15]: [Training] loss = 0.002074 * 110, metric = 0.00% * 110 3.158s ( 34.8 samples/s);
Finished Epoch[11 of 15]: [Training] loss = 0.001020 * 110, metric = 0.00% * 110 3.160s ( 34.8 samples/s);
Finished Epoch[12 of 15]: [Training] loss = 0.000564 * 110, metric = 0.00% * 110 3.146s ( 35.0 samples/s);
Finished Epoch[13 of 15]: [Training] loss = 0.003924 * 110, metric = 0.00% * 110 3.133s ( 35.1 samples/s);
Finished Epoch[14 of 15]: [Training] loss = 0.000684 * 110, metric = 0.00% * 110 3.131s ( 35.1 samples/s);
Finished Epoch[15 of 15]: [Training] loss = 0.000755 * 110, metric = 0.00% * 110 3.148s ( 34.9 samples/s);
```

Table 4.4: Training process of auger_34_old model for 15 epochs.

and Figure 4.28 show the details of training process. After training the model we tested it with our test set of 55 images from csa_ramp.wav and obtained results as shown in Figure 4.29. We further tested its performance with 55 random images of audio from powder product psa_ramp.wav and obtained results as shown in Figure 4.30. So in Figure 4.29, you can see it predicted 51 images correctly and labeled 4 images wrongly and for powder product you can see in the Figure 4.30 it predicted 47 images correctly and 8 images wrongly.

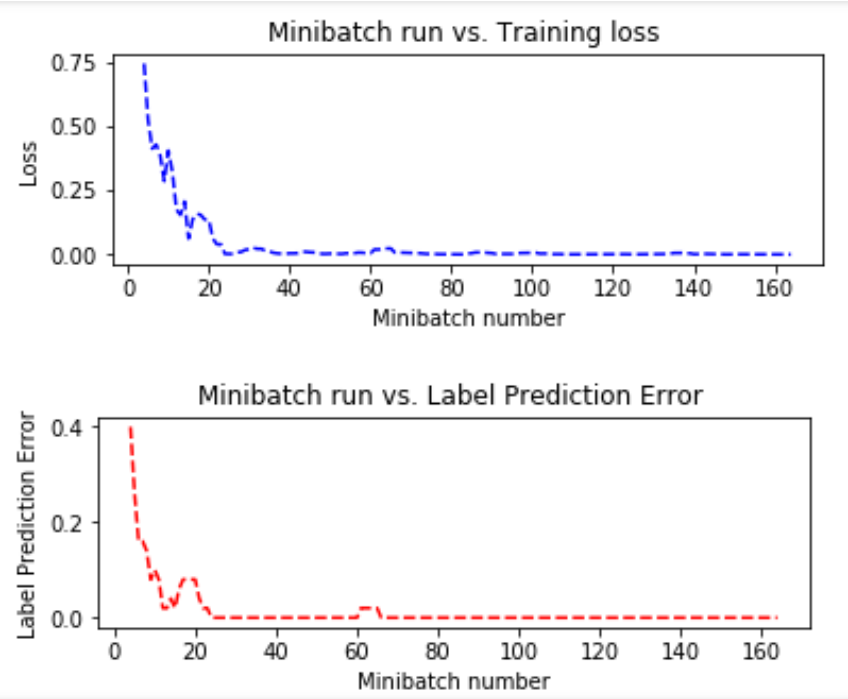


Figure 4.28: Training graph of auger_34_old model.

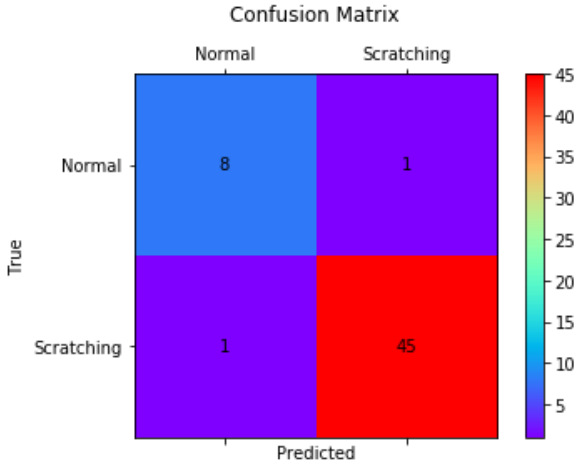


Figure 4.29: Testing, 53/55 test images predicted correctly from **coscus** csa_ramp.wav.

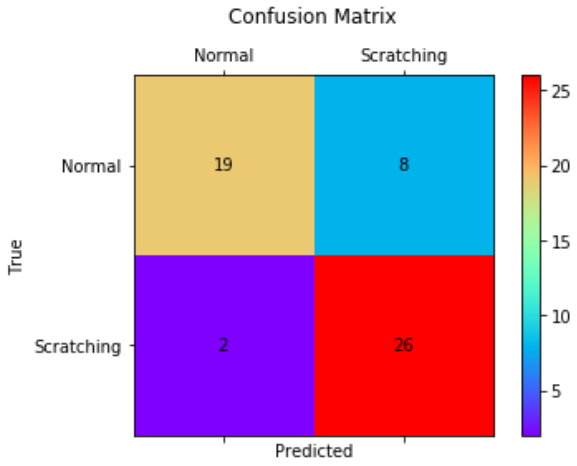


Figure 4.30: Testing, 45/55 test images predicted correctly from **powder** psa_ramp.wav.

4.7 Summarizing results

In our experiments, we explored the answers to our two questions discussed in section 4.1. First, we collected all the data using different sensors and then performed frequency domain analysis to figure out scratching frequencies. We observed some new frequencies (9.5k to 13.5 kHz) at high rotating speeds. Hence, we labeled our data accordingly for "Normal" and "Scratching" sound in time domain. Secondly, we converted these audio signals in to a series of spectral images and trained ResNet architecture based deep models on these images. We tested these models on two different products "Coscus" and "Powder" and documented our results. Following Table 4.5 shows the overall summary of our experimental results. We can see that 18-layer model achieved 90% accuracy and 34-layer model performed slightly better with 91% accuracy.

Safe limit for rpm		
	Coscus (rpm)	Powder (rpm)
New	574	662
Old	221	165

Test results of 18-layer model			
	Coscus	Powder	Overall accuracy
New	53/55	49/55	200/220
Old	51/55	47/55	= 90.9%

Test results of 34-layer model			
	Coscus	Powder	Overall accuracy
New	54/55	49/55	201/220
Old	53/55	45/55	= 91.3 %

Table 4.5: Summarizing experimental results for New and Old auger with Coscus and Powder products.

Chapter 5

Future work

In this thesis, we have presented a potential solution to auger scratching problem. First, we explored the safe operation limits of machine for different products by doing frequency domain analysis on audio signals. Then we automated the task of anomaly (scratching) detection by using a state of the art machine learning models. On industrial level, timely detection and reporting of anomalies is very important especially when there are multiple machines operating at the same time. Following Figure 5.1 presents an overall architecture in an industrial setting. In a typical machine room in an industrial environment, there are

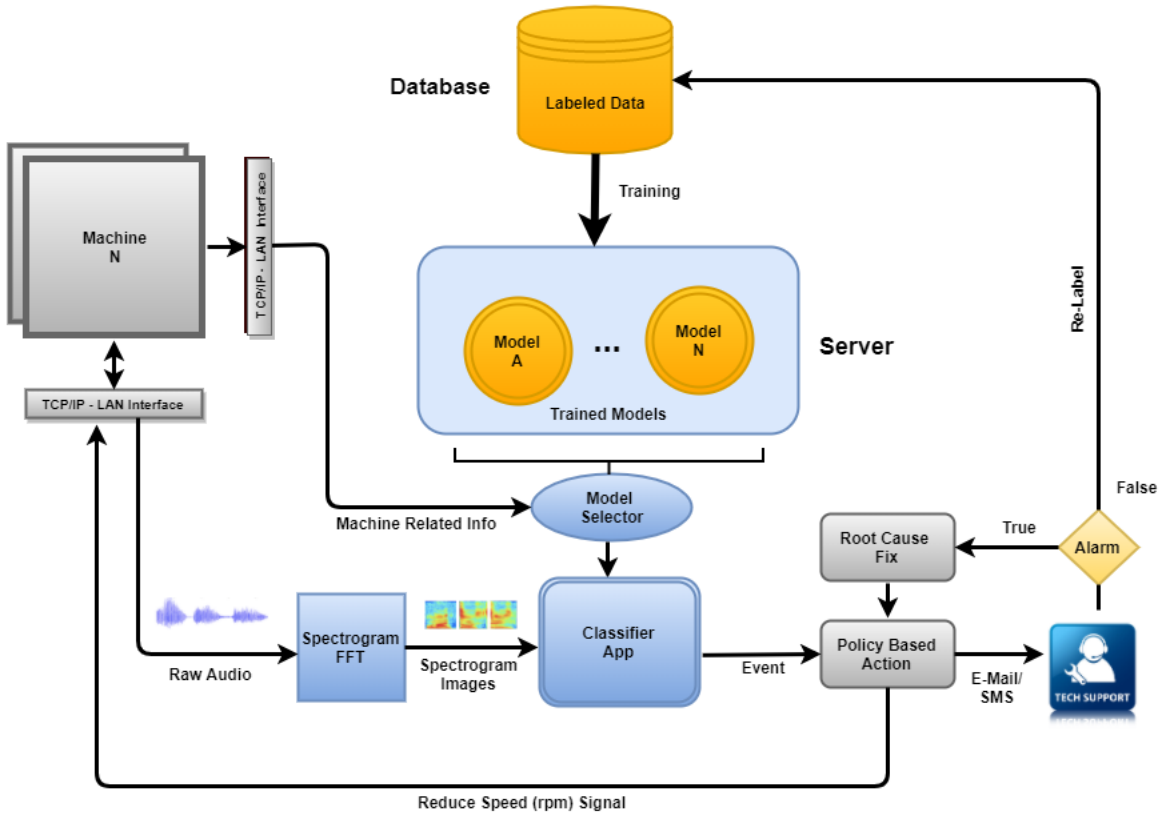


Figure 5.1: Overall architecture in industrial setting.

many machines operating at the same time. As you can see in Figure 5.1, we can collect audio signals from multiple machines and transport them over a LAN to a local server. Typically, on a server there are multiple pre-trained models running for each machine. We can select a relevant model in a classifier app and classify the audio signal based on spectrogram images for "Normal" and "Scratching" states. In case of scratching, we can

generate an event and send a “Reduce Speed/Stop” signal back to relevant machine. We can further generate and E-Mail or SMS to Tech Support team based on our company policy. Tech Support team can analyze the problem and fix it as soon as possible. If they find out that it was a false alarm, they can relabel the data and update the database to retrain the relevant model on the basis of new data. This way we can not only automate the entire process but also create an active learning environment.

5.1 Further applications

The solution we presented here is based on visual anomalies and can be easily extended to other domains as well. For example, we can use the same technique in cars to detect abnormalities in engine such as “Knocking” and “Ticking” sounds. We can use it to detect blown up or overloaded motors by detecting abnormalities in sound. We can use this technique in medical industry to detect abnormalities in x-rays of bone e.g. broken bones.

Bibliography

[1] Bong Jun Ko, Jorge Ortiz, Theodoros Salonidis, Maroun Touma, Dinesh Verma, Shiqiang Wang, Xiping Wang, and David Wood. Acoustic signal processing for anomaly detection in machine room environments: Demo abstract. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*, pages 213–214. ACM, 2016.

[2] Francesco Camastra, Alessandro Vinciarelli, and Jie Yu. Machine learning for audio, image and video analysis. *J. Electronic Imaging*, 18(2):029901, 2009.

[3] Alexander Ypma. Learning methods for machine vibration analysis and health monitoring. 2001. URL http://rduin.nl/papers/thesis_01_ypma.pdf.

[4] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[7] Yoshua Bengio. Deep learning of representations for unsupervised and transfer learning. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, pages 17–36, 2012.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[9] Michael Nielsen. Using neural nets to recognize handwritten digits, 2015. http://www.math.hkbu.edu.hk/~mhyipa/nndl/chap1.pdf.

[10] Andrew Ng. Deep learning cs229, 2017. http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf.

[11] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[12] Andrew Senior, Georg Heigold, Ke Yang, et al. An empirical study of learning rates in deep neural networks for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6724–6728. IEEE, 2013.

[13] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[14] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[15] Jianxin Wu. Introduction to convolutional neural networks. *National Key Lab for Novel Software Technology. Nanjing University. China*, 2017.

[16] Haibing Wu and Xiaodong Gu. Max-pooling dropout for regularization of convolutional neural networks. *arXiv preprint arXiv:1512.01400*, 2015.

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998.

[18] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[19] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

[20] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pages 2924–2932, 2014.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *arXiv preprint arXiv:1603.05027*, 2016.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. 2015. *arXiv preprint arXiv:1502.01852*.

[24] Aaditya Pise. Machine learning concept for the premature recognition of machine faults using vibration and specialized body sound sensors, 2016.

[25] Chanchal Kumar. Pulse code modulation, lecture slides, 2017. http://ecehithaldia.in/teaching_material/Pulse%20Code%20Modulation_JB336515629.pdf.

[26] C Sidney Burrus. *Fast fourier transforms*. Lulu. com, 2008.

[27] National Instruments. Understanding ffts and windowing, 2017. http://download.ni.com/evaluation/pxi/Understanding%20FFTs%20and%20Windowing.pdf.

[28] Jerome Sueur. A very short introduction to sound analysis for those who like elephant trumpet calls or other wildlife sound, 2016. https://cran.r-project.org/web/packages/seewave/vignettes/seewave_analysis.pdf.

[29] Team Microsoft. The microsoft cognitive toolkit, 2016. https://docs.microsoft.com/en-us/cognitive-toolkit/.

[30] Christopher Olah. Calculus on computational graphs: Backpropagation. *Colah's Blog*, 2015. http://colah.github.io/posts/2015-08-Backprop/.

[31] Team Microsoft. Cntk packages, 2016. https://docs.microsoft.com/en-us/python/api/cntk?view=cntk-py-2.3.

List of Figures

List of Tables