

CLASSIFICATION OF SIGNALS USING CONVOLUTIONAL NEURAL NETWORK

PROJECT REPORT

MASTERS OF SCIENCE

(Honors School)

IN

PHYSICS (Specialization in Electronics)

Jan-May 2019




Submitted by

**LAKSHITA RAWAT
SWAGATA BANERJEE**

Under the supervision of

Prof. NAVDEEP GOYAL


Signature of Supervisor

DEPARTMENT OF PHYSICS

PANJAB UNIVERSITY

CHANDIGARH-160014

ACKNOWLEDGEMENT

“Just as a spring breeze awakens tender new shoots of green, sincere encouragement can thaw a frozen heart and instill courage. It is the most powerful mean to rejuvenate the human spirit.” (Dr. Daisaku Ikeda) We take this opportunity to offer our sincere gratitude to all who could impart us with such encouragement which lead us to courageously explore during this project.

Heartiest gratitude to our project supervisor **Prof. Navdeep Goyal** (Chairman, Physics department, Panjab University) for his trust in us throughout our endeavors.

We sincerely thank **Dr. Gurpratap Singh** (UIET Department, Panjab University) for his genuine guidance.

The project would not have beared any fruit without the expert guidance of **Mr. Gaurav Kumar** (Director, Magma Research and Consultancy Services) and **Prof. Ajay Deogar** (NITTTR, Chandigarh). We offer our deepest gratitude to both of them for their constant support throughout the work.

“Each form of life supports all others; together they weave the grand web of life.” (Dr. Daisaku Ikeda). From the spark of idea of the aim of project, through the struggles, to the end, **Mr. Rohan Sahni** (Student, M.Sc. Physics, Panjab University) has been generous to work in the team as our co-partner with his unwavering commitment. We feel fortunate and offer our heartiest gratitude for his contribution.

We are ever more grateful to our parents for their faith in us, to our faculties and friends in Masters’ class for their support and smiles which kept us motivated.

We started with a curiosity and now our spirits are revived because of such encouragement. We stand more curious to explore greater possibilities to contribute better to human life and its enrichment. This result and our future endeavors based on it, is our token of gratitude to all of you.

ABSTRACT

According to WHO, heart disease and stroke are the world's biggest killers, accounting for a combined 15.2 million deaths in 2016. These diseases have remained the leading causes of death globally in the last 15 years.

In this project we first study CNN which is one of the Deep learning methods. Then, we review the Paper presented by Kachuee. On the basis of this, we then repeat the mentioned experiment of ECG classification using python (Keras library) and test variation of optimizer and hence see the accuracy.

Electrocardiogram (ECG) can be reliably used as a measure to monitor the functionality of the cardiovascular system. Recently, there has been a great attention towards accurate categorization of heartbeats, which could be an asset in heart disease risk detection. While there are many commonalities between different ECG conditions, the focus of most studies has been classifying a set of conditions on a dataset annotated for that task. In this project, we have reviewed a method based on deep convolutional neural networks for the classification of heartbeats (ECGs) which is able to accurately classify five different arrhythmias in accordance with the AAMI EC57 standard.

We evaluated the proposed method on PhysionNet's MIT-BIH dataset. According to the results, the suggested method was able to make predictions with the good average accuracies on arrhythmia classification. The precision was further checked with different optimizer in the model. The average accuracies were then reported for both and compared.

Table of Contents

ACKNOWLEDGEMENT	2
ABSTRACT	3
INTRODUCTION.....	5
1. CNN	6
1.1 First Layer (Convolution).....	6
1.2 Non Linearity (ReLU) layer.....	9
1.3 The Pooling layer	10
1.3.1 Story so far:.....	12
1.4 Fully Connected Layer.....	12
1.5 Putting it all together – Training using Backpropagation.....	13
1.6 Testing.....	16
1.7 Now we very well understand that for a neural network, we need:	16
2. EXPERIMENT	17
2.1 Aim:.....	17
2.2 Dataset.....	17
2.2.1 Methodology	17
2.2 Architecture of the proposed network	19
2.2.1 Training the Arrhythmia Classifier.....	19
2.3 Loss function used	20
2.4 Optimization.....	20
2.5 Evaluation criteria	22
To note	22
2.6 Python Code 1 {using Optimizer:- 1)Adam 2)SGD }	23
3. RESULT	34
4. CONCLUSION.....	34
5. Future Prospect That We Wish to Undertake	34
6. References.....	35

INTRODUCTION

“Intelligence is the ability to adapt to changes” (Stephen Hawking). This ability to adapt can be said to be, the ability to ‘acquire and apply’: knowledge and skills. Learning, intuition, creativity and inference are some tasks associated with intelligence. AI is a quest to model and implement theories of intelligence. “Artificial Intelligence (AI) is an area of computer science concerned with designing intelligent computer systems” that is, systems that exhibit the characteristics we associate with intelligence in human behaviour” (Avron Barr and Feigenbaum, 1981). One of the technologies for such intelligent problem solving is Neural Networks.

Neural networks are motivated from the computing performed by a human brain. Replicating the nature of human brain, NN is an interconnected network of processing units called neurons. NN learns by examples, same as our brain. It trains itself with known examples, to ‘acquire’ knowledge about these. Once efficiently trained the system then can predict or solve problem instances, ‘applying’ the knowledge acquired to use.

NN uses learning mechanisms which can be supervised or unsupervised. In supervised learning a teacher presents desired outputs and the NN system minimizes the error between computed output and the target output presented by teacher to achieve better accuracy. While in unsupervised learning there is no teacher, the network tries to learn by itself, organizing the input instances of the problem.

Several NN architectures have evolved since time. Some of the well known architectures include backpropagation, perceptron, etc. In this work we will study and implement CNN architecture to classify specific signals.

1. CNN

How does a human brain recognize/classify an image/signal? When it sees an image, some neurons are fired when exposed to vertical edges and some when to horizontal or diagonal edges. Hubel and Wiesel found out that all of these neurons were organized in a columnar architecture and that together, they were able to produce visual perception. This idea of specialized components inside of a system having specific tasks (the neuronal cells in the visual cortex looking for specific characteristics) is one that machines use as well, and is the basis behind CNNs.

There are four main operations in the Convolutional Network:-

1. Convolution
2. Non Linearity (ReLU)
3. Pooling or Sub Sampling
4. Classification (Fully Connected Layer)

These operations are the basic building blocks of *every* Convolutional Neural Network, so understanding how these work is an important step to developing a sound understanding of ConvNets. We will try to understand the intuition behind each of these operations below.

1.1 First Layer (Convolution)

The first layer in a CNN is always a Convolutional Layer. The input to this layer is the sample (image, signal etc.) which is usually taken as a $32 \times 32 \times N$ array of pixel values (N being 3 in RGB image otherwise 1 in gray scale image or 1D data). Now, the best way to explain a convolutional layer is to imagine a flashlight that is shining over the top left of the image. Let's say that the light this flashlight shines covers a 5×5 area. And now, let's imagine this flashlight sliding across all the areas of the input image. In machine learning terms, this flashlight is called a filter (or sometimes referred to as a neuron or a kernel) which is also an array of numbers (the numbers are called weights or parameters). This filter shines over an array of values (pixels values or amplitude) of the image or data sample. The region that it is shining over is called the receptive field. Important note is that the depth of this filter has to be the same as the depth of the input. The dimensions of this filter may be like $5 \times 5 \times n$. The filter is convolved over one part of the sample starting from the top left of the image or sample. It then slides with the fixed amount (called stride) across the input sample. Convolution in each stride, it is multiplying the values in the filter with the original pixel values of the image/sample. (Aka computing element wise multiplications). These multiplications are all summed up. So we have a single number from the convolution of the filter to a single receptive region on the sample. Remember, this number is the response of the filter to that receptive region. We repeat this process, for every location on the input

volume. (Next step would be moving the filter to the right by 1 stride and so on). Every unique location on the input volume produces a number. After sliding the filter over all the locations, you will find out that what you're left with is a $28 \times 28 \times 1$ array of numbers, which we call an activation map or feature map. The reason you get a 28×28 array is that there are 784 different locations that a 5×5 filter can fit on a 32×32 input image. These 784 numbers are mapped to a 28×28 array.

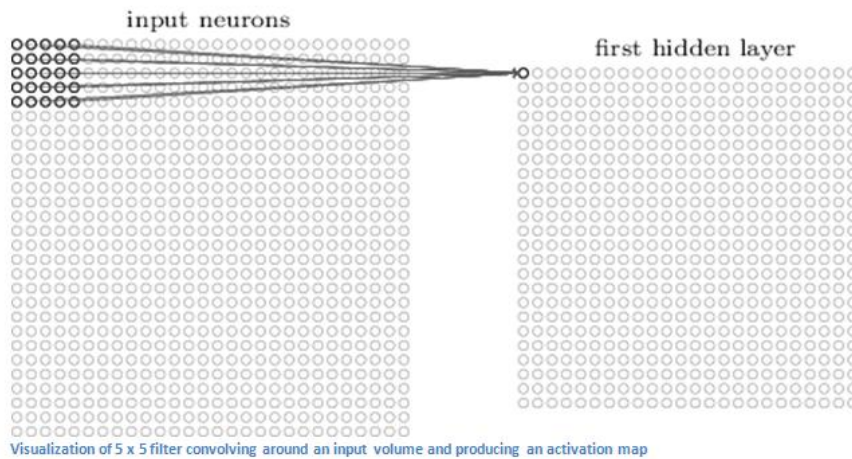


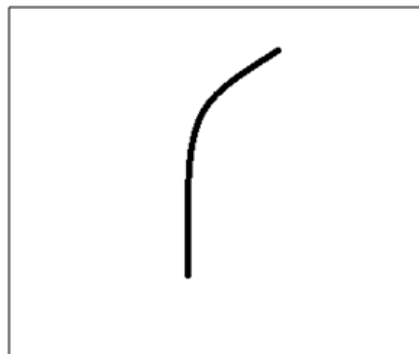
Figure 1

("Neural Networks and Deep Learning" by Michael Nielsen.)

Each of these filters can be thought of as feature identifiers like straight edges, curves etc.

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter

Figure 2

Now let's take an example of an image that we want to classify, and let's put our filter at the top left corner. Remember, what we have to do is multiply the values in the filter with the original pixel values of the image.

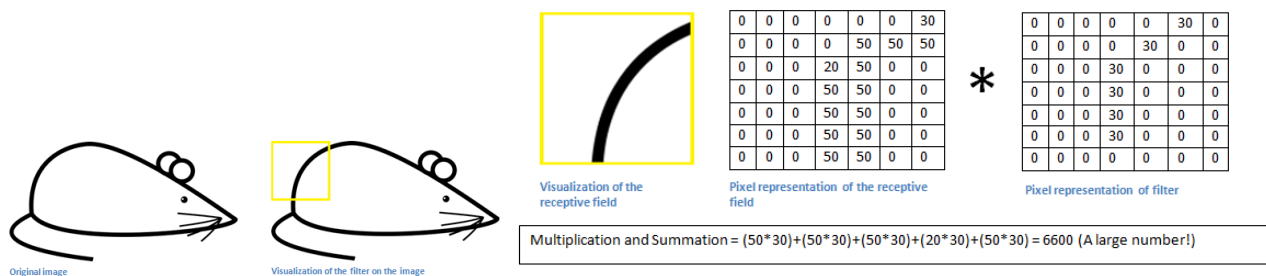


Figure 3

Basically, in the input image, if there is a shape that generally resembles the curve that this filter is representing, then all of the multiplications summed together will result in a large value! Now let's see what happens when we move our filter.

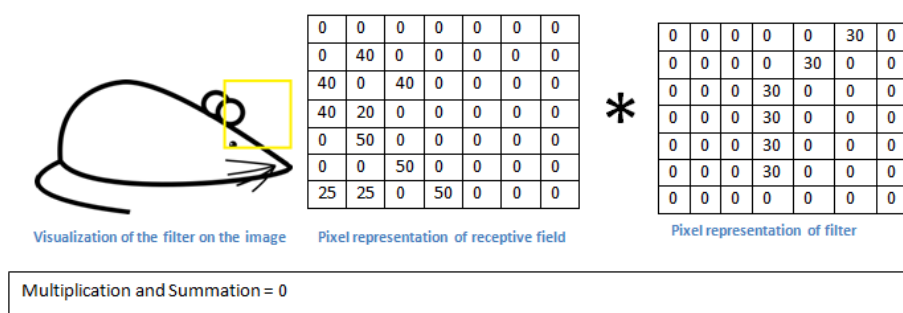


Figure 4

The value is lower when there is nothing in the image section that responded to the curve detector filter. The output of this convolutional layer is an activation map. So, in the simple case of a one filter convolution (and if that filter is a curve detector), the activation map will show the areas in which there are mostly likely to be curves in the picture. In this example, the top left value of our $26 \times 26 \times 1$ activation map (26 because of the 7×7 filter instead of 5×5) will be 6600. This high value means that it is likely that there is some sort of curve in the input volume that caused the filter to activate. The top right value in our activation map will be 0 because there isn't anything in the input volume that caused the filter to activate (or more simply said, there wasn't a curve in that region of the original image). Remember, this is just for one filter. This is just a filter that is going to detect lines that curve outward and to the right. We can have other filters for lines that curve to the left or for straight edges. The more filters, the greater the depth of the activation map, and the more information we have about the input volume.

The size of the Feature Map (Convolved Feature) is controlled by three parameters that we need to decide before the convolution step is performed:

- A) Depth: Depth corresponds to the number of filters we use for the convolution operation. In the network shown in Figure 7, we are performing convolution of

the original boat image using three distinct filters, thus producing three different feature maps as shown. You can think of these three feature maps as stacked 2d matrices, so, the ‘depth’ of the feature map would be three.

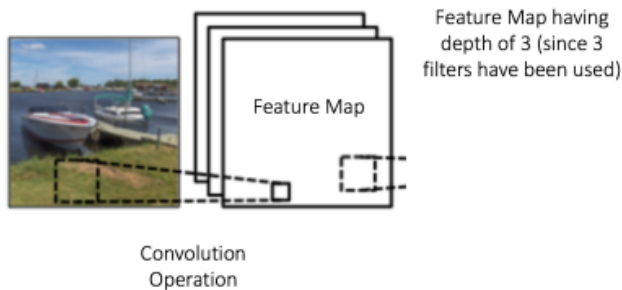


Figure 5

- B) Stride: Stride is the number of pixels by which we slide our filter matrix over the input matrix. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2, then the filters jump 2 pixels at a time as we slide them around. Having a larger stride will produce smaller feature maps.
- C) Zero-padding: Sometimes, it is convenient to pad the input matrix with zeros around the border, so that we can apply the filter to bordering elements of our input image matrix. A nice feature of zero padding is that it allows us to control the size of the feature maps. Adding zero-padding is also called *wide convolution*, and not using zero-padding would be a *narrow convolution*. This has been explained clearly in.

1.2 Non Linearity (ReLU) layer

An additional operation called ReLU has been used after every Convolution operation. ReLU stands for Rectified Linear Unit and is a non-linear operation. Its output is given by:

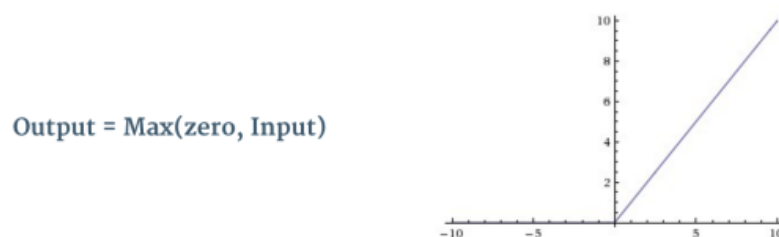


Figure 6: ReLU Operation

ReLU is an element wise operation (applied per pixel/data) and replaces all negative pixel/data values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in our ConvNet, since most of the real-world data we would want our ConvNet to learn would be non-linear (Convolution is a linear operation – element wise matrix multiplication and addition, so we account for non-linearity by introducing a non-linear

function like ReLU). The ReLU operation can be understood clearly from figure 7 below. It shows the ReLU operation applied to one of the feature maps of an image after first convolutional layer. The output feature map here is also referred to as the ‘Rectified’ feature map. Other nonlinear functions such as tanh or sigmoid can also be used instead of ReLU, but ReLU has been found to perform better in most situations.

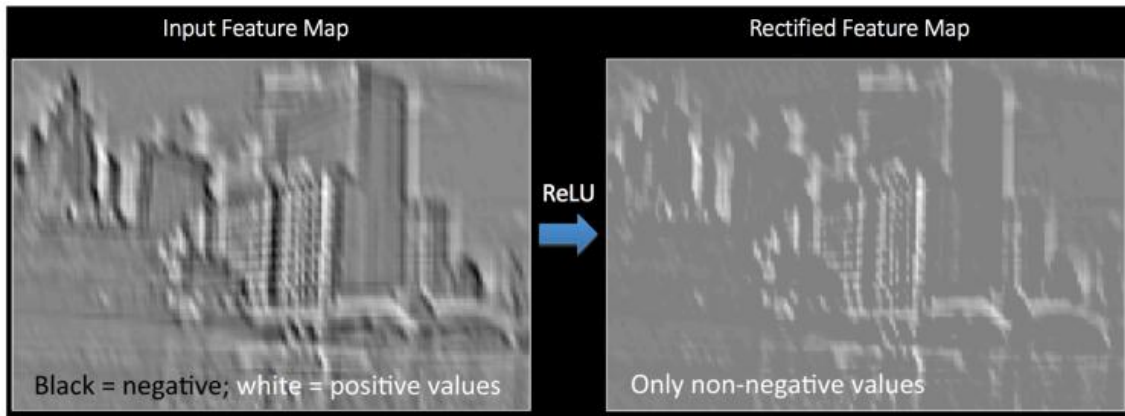


Figure 7

1.3 The Pooling layer

Spatial Pooling (also called subsampling or down sampling) reduces the dimensionality of each feature map but retains the most important information. Spatial Pooling can be of different types: Max, Average, Sum etc.

In case of Max Pooling, we define a spatial neighborhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average (Average Pooling) or sum of all elements in that window. In practice, Max Pooling has been shown to work better.

Figure 8 shows an example of Max Pooling operation on a Rectified Feature map (obtained after convolution + ReLU operation) by using a 2×2 window.

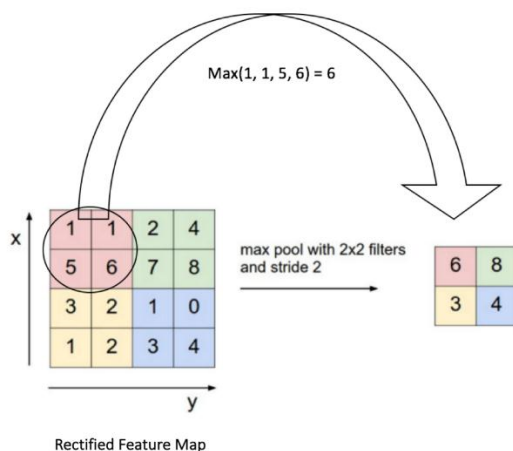


Figure 8: Max Pooling

Here, we slide our 2 x 2 window by 2 cells (also called ‘stride’) and take the maximum value in each region. As shown in Figure, this reduces the dimensionality of our feature map. In the network shown in Figure below, pooling operation is applied separately to each feature map (notice that, due to this, we get three output maps from three input maps).

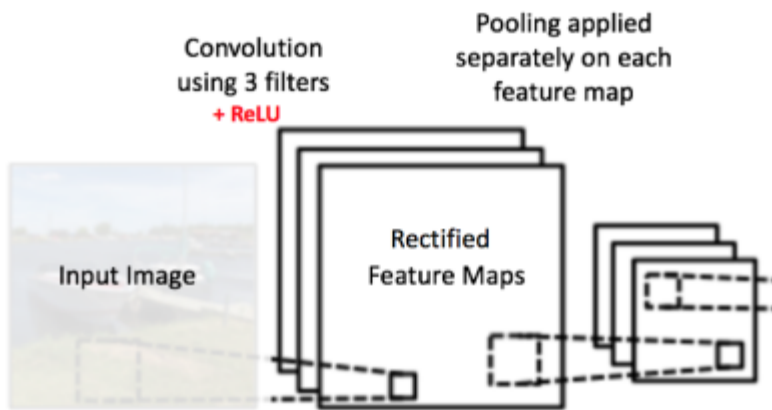


Figure 9: Pooling applied to Rectified Feature Maps

Figure 9 shows the effect of Pooling on the Rectified Feature Map we received after the ReLU operation in Figure above.

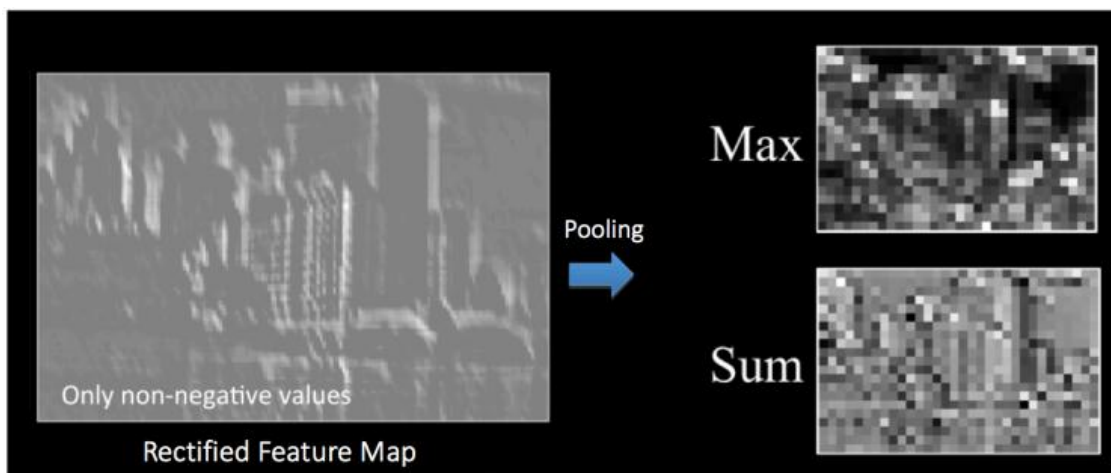


Figure 10: Pooling.

Pooling layer:-

Progressively reduces the spatial size of the input representation [4], making the input representations (feature dimension) more manageable.

Reduces the number of parameters and computations in the network, therefore, controlling over fitting.

Makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling – since we take the maximum / average value in a local neighborhood).

Helps us arrive at an almost scale invariant representation of our image (the exact term is “equivariant”). This is very powerful since we can detect objects in an image no matter where they are located.

1.3.1 Story so far:

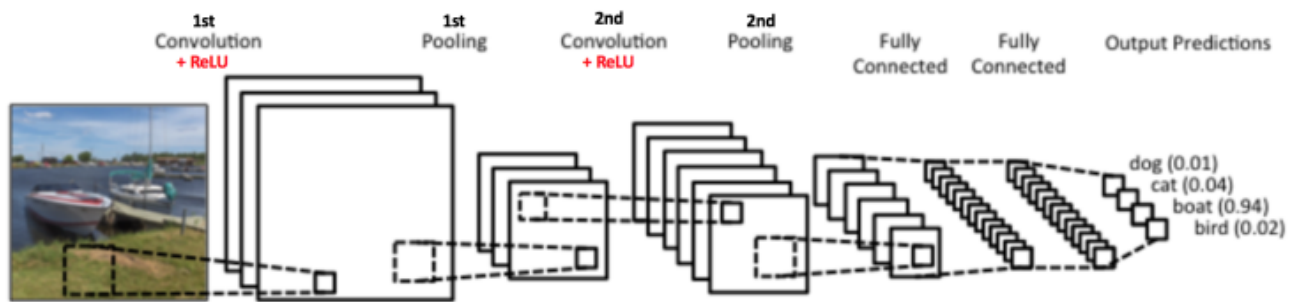


Figure 11

So far we have seen how Convolution, ReLU and Pooling work. It is important to understand that these layers are the basic building blocks of any CNN. As shown in Figure, we have two sets of Convolution, ReLU & Pooling layers – the 2nd Convolution layer performs convolution on the output of the first Pooling Layer using six filters to produce a total of six feature maps. ReLU is then applied individually on all of these six feature maps. We then perform Max Pooling operation separately on each of the six rectified feature maps.

Together these layers extract the useful features from the images, introduce non-linearity in our network and reduce feature dimension while aiming to make the features somewhat equivariant to scale and translation. Here, the output of the 2nd Pooling Layer acts as an input to the Fully Connected Layer.

1.4 Fully Connected Layer

The Fully Connected layer is a traditional Multilayer Perceptron that uses a softmax activation function in the output. The term “Fully Connected” implies that every neuron in the previous layer is connected to every neuron on the next layer.

The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of the Fully Connected layer is to use these features for classifying the input image into various classes based on the training dataset. For example, the image classification task we set out to perform has four possible outputs as shown in figure 12 below (note that Figure 14 does not show connections between the nodes in the fully connected layer)

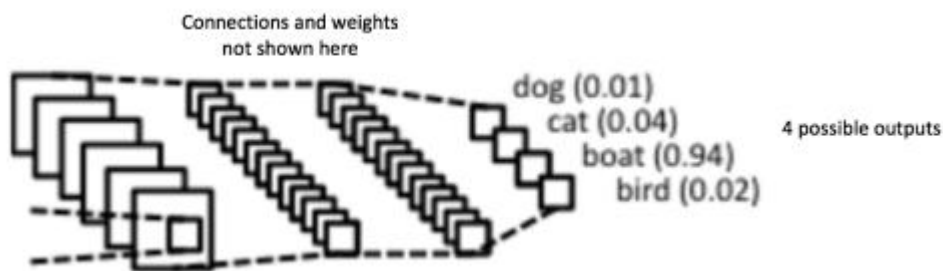


Figure 12: Fully Connected Layer -each node is connected to every other node in the adjacent layer

The sum of output probabilities from the Fully Connected Layer is 1. This is ensured by using the softmax as the activation function in the output layer of the Fully Connected Layer. The Softmax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one.

1.5 Putting it all together – Training using Back propagation

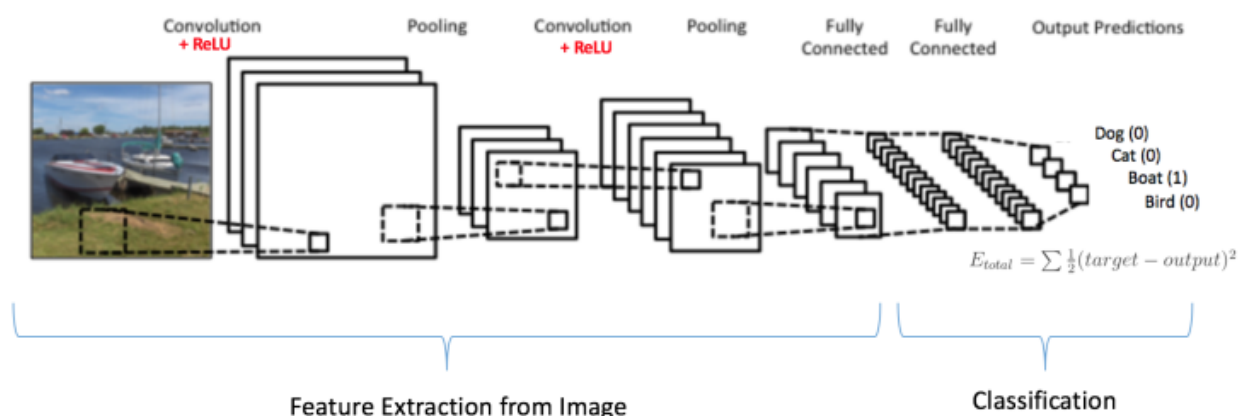


Figure 13: Training the ConvNet

The overall training process of the Convolution Network may be summarized as below:-

Step1: We initialize all filters and parameters / weights with random values

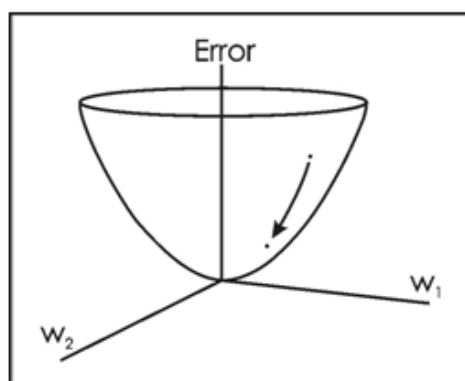
Step2: The network takes a training image as input, goes through the forward propagation step (convolution, ReLU and pooling operations along with forward propagation in the Fully Connected layer) and finds the output probabilities for each class. Let's say the output probabilities for the boat image above are [0.2, 0.4, 0.1, 0.3] since weights are randomly assigned for the first training example, output probabilities are also random.

Step3: Calculate the total error at the output layer (summation over all 4 classes). The network, with its current weights, isn't able to look for those low level features or thus isn't able to make any reasonable conclusion about what the classification might be. This goes to the loss function part of back propagation. Remember that what we are using right now is training data. This data has both an image and a label. Let's say for example that the first training image inputted was a 3. The label for the image would be

[0 0 0 1 0 0 0 0 0]. A loss function can be defined in many different ways but a common one is MSE (mean squared error), which is $\frac{1}{2}$ times (actual - predicted) squared.

$$E_{total} = \sum \frac{1}{2} (target - output)^2$$

Let's say the variable L is equal to that value. As you can imagine, the loss will be extremely high for the first couple of training images. Now, let's just think about this intuitively. We want to get to a point where the predicted label (output of the ConvNet) is the same as the training label (This means that our network got its prediction right). In order to get there, we want to minimize the amount of loss we have. Visualizing this as just an optimization problem in calculus, we want to find out which inputs (weights in our case) most directly contributed to the loss (or error) of the network.



One way of visualizing this idea of minimizing the loss is to consider a 3-D graph where the weights of the neural net (there are obviously more than 2 weights, but let's go for simplicity) are the independent variables and the dependent variable is the loss. The task of minimizing the loss involves trying to adjust the weights so that the loss decreases. In visual terms, we want to get to the lowest point in our bowl shaped object. To do this, we have to take a derivative of the loss (visual terms: calculate the slope in every direction) with respect to the weights.

Figure 14

This is the mathematical equivalent of a dL/dW where W are the weights at a particular layer. Now, what we want to do is perform a backward pass through the network, which is determining which weights contributed most to the loss and finding ways to adjust them so that the loss decreases. Once we compute this derivative, we then go to the last step which is the weight update. This is where we take all the weights of the filters and update them so that they change in the opposite direction of the gradient.

$$w = w_i - \eta \frac{dL}{dW}$$

w = Weight
w_i = Initial Weight
η = Learning Rate

The learning rate is a parameter that is chosen by the programmer. A high learning rate means that bigger steps are taken in the weight updates and thus, it may take less time for the model to converge on an optimal set of weights. However, a learning rate that is too high could result in jumps that are too large and not precise enough to reach the optimal point.

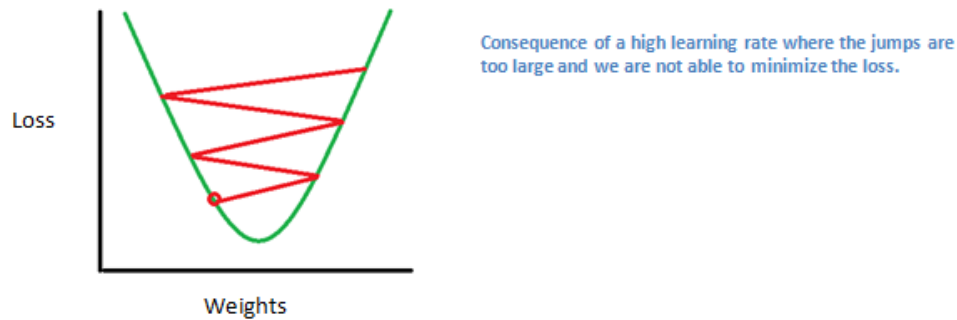


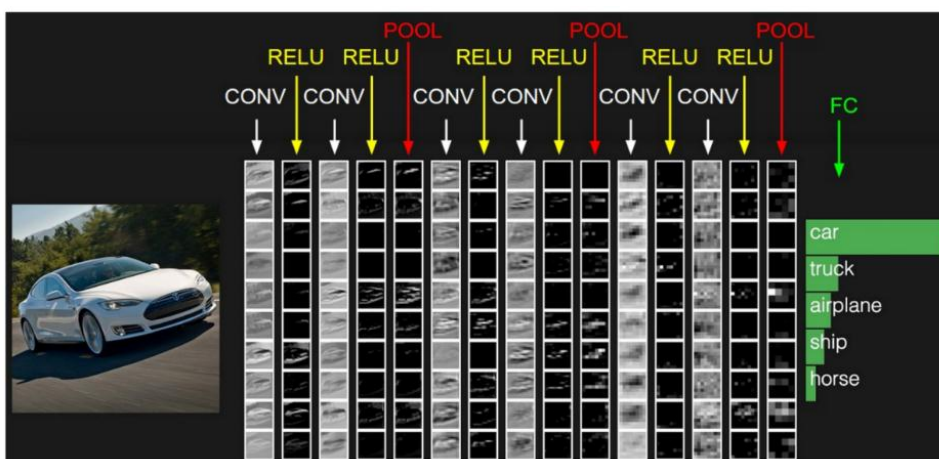
Figure 15

The process of forward pass, loss function, backward pass, and parameter update is one training iteration. The program will repeat this process for a fixed number of iterations for each set of training images (commonly called a batch). Once you finish the parameter update on the last training example, hopefully the network should be trained well enough so that the weights of the layers are tuned correctly.

Step4: Use Back propagation to calculate the gradients of the error with respect to all weights in the network and use gradient descent to update all filter values / weights and parameter values to minimize the output error. The weights are adjusted in proportion to their contribution to the total error.

Step5: Repeat steps 2-4 with all images in the training set. The above steps train the ConvNet – this essentially means that all the weights and parameters of the ConvNet have now been optimized to correctly classify images from the training set.

When a new (unseen) image is input into the ConvNet, the network would go through the forward propagation step and output a probability for each class (for a new image, the output probabilities are calculated using the weights which have been optimized to correctly classify all the previous training examples). If our training set is large enough, the network will (hopefully) generalize well to new images and classify them into correct categories.



1.6 Testing

Finally, to see whether or not our CNN works, we have a different set of images and labels (can't double dip between training and test!) and pass the images through the CNN. We compare the outputs to the ground truth and see if our network works!

1.7 Now we very well understand that for a neural network, we need:

1. Data (training and testing)
2. Model: A model architecture, Arrays of weight vectors, Arrays of bias vectors, an activation function.
3. Evaluation: A function to convert the output to a probability distribution.
4. A loss function.
5. Optimization: A way to use the output of the model to tune the model iteratively and improve it (error back-propagation).

2. EXPERIMENT¹

2.1 Aim:

Classification of heartbeats (ECG Signals) of 5 different arrhythmias in accordance with AAMI EC57 standard using Convolutional neural network.

Category	Annotations
N	<ul style="list-style-type: none">• Normal• Left/Right bundle branch block• Atrial escape• Nodal escape
S	<ul style="list-style-type: none">• Atrial premature• Aberrant atrial premature• Nodal premature• Supra-ventricular premature
V	<ul style="list-style-type: none">• Premature ventricular contraction• Ventricular escape
F	<ul style="list-style-type: none">• Fusion of ventricular and normal
Q	<ul style="list-style-type: none">• Paced• Fusion of paced• Unclassifiable

2.2 Dataset

PhysioNet MIT-BIH Arrhythmia ECG Databases has been used as data source for labeled ECG records. The MIT-BIH dataset consists of ECG recordings from 47 different subjects recorded at the sampling rate of 360Hz which were resampled at 125Hz. Each beat was annotated by at least two cardiologists. We use annotations in this dataset to create five different beat categories in accordance with Association for the Advancement of Medical Instrumentation (AAMI) EC57 standard. See above Table for a summary of mappings between beat annotations in each category.

2.2.1 Methodology

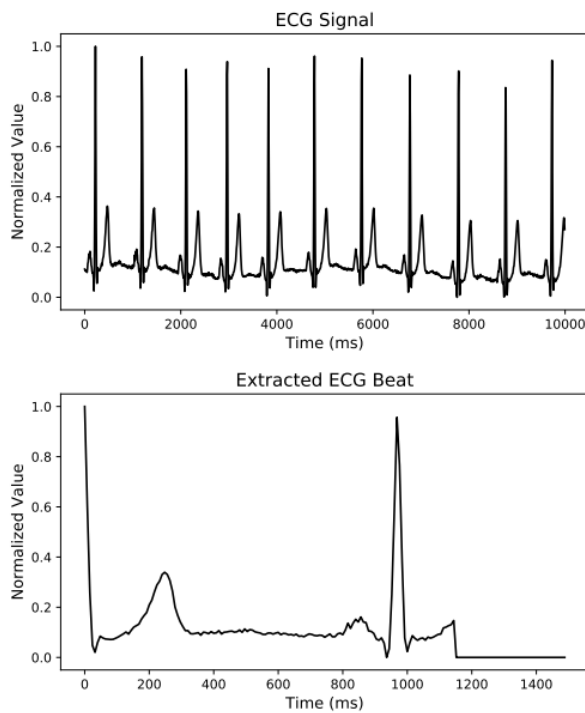
Preprocessing:

As ECG beats are inputs of the proposed method we suggest a simple and yet effective method for preprocessing ECG signals and extracting beats. The steps used for extracting beats from an ECG signal are as follows:-

1. Splitting the continuous ECG signal to 10s windows and select a 10s window from an ECG signal.

¹

2. Normalizing the amplitude values to the range of between zero and one.
3. Finding the set of all local maximums based on zero- crossings of the first derivative.
4. Finding the set of ECG R-peak candidates by applying a threshold of 0.9 on the normalized value of the local maximums.
5. Finding the median of R-R time intervals as the nominal heartbeat period of that window (T).
6. For each R-peak, selecting a signal part with the length equal to $1.2T$.
7. Padding each selected part with zeros to make its length equal to a predefined fixed length.²



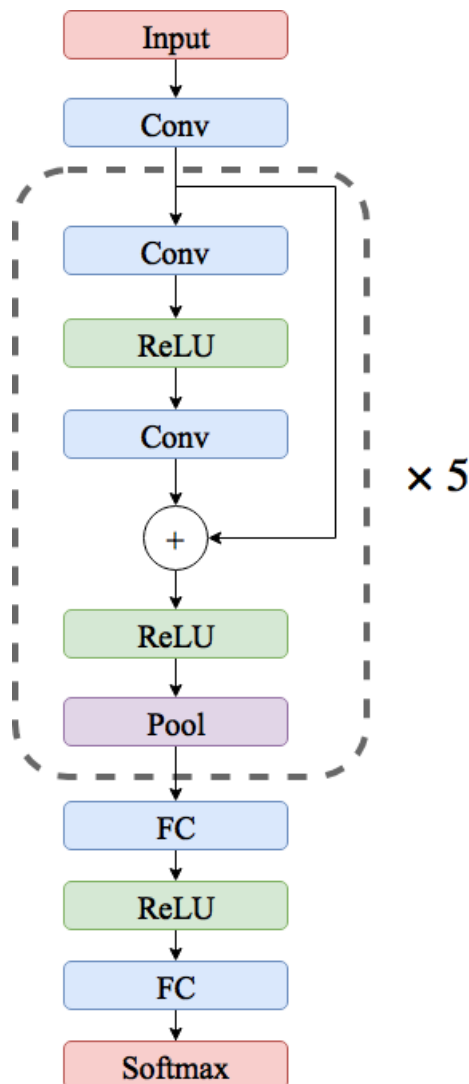
It is worth mentioning that the suggested beat extraction method is simple and effective in extracting R-R intervals from signals with different morphologies. For example, we have not used any form of filtering or any processing that makes any assumption about the signal morphology or spectrum. Additionally, all the extracted beats have identical lengths which is essential for being used as inputs to the subsequent processing parts.²

The signals are stored as arrays in .csv format. 187 columns defined 187 discrete values of each signal. Training data file has 87554 signals, while test data has 21892 signals pertaining to 5 categories described above.

¹ Kachuee, M. &.-4. (n.d.). ECG Heartbeat Classification: A deep transferable Representation.

2.2 Architecture of the proposed network

¹Fig. below illustrates the network architecture proposed for the beat classification task.



2.2.1 Training the Arrhythmia Classifier

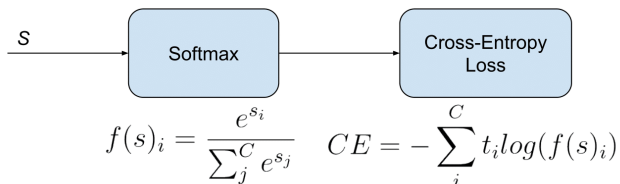
Training of a convolutional neural network is done for classification of ECG beat types on the MIT-BIH dataset. The trained network is used for the purpose of beat classification.

- Here, all convolution layers are applying 1-D convolution through time and each have 32 kernels/filters of size 5.
- We also use max pooling of size 5 and stride 2 in all pooling layers.
- The predictor network consists of five residual blocks followed by two fully-connected layers with 32 neurons each and a softmax layer to predict output class probabilities.
- Each residual block contains two convolutional layers, two ReLU nonlinearities, a residual skip connection and a pooling layer.

- In total, the resulting network is a deep network consisting of 13 weight layers.

2.3 Loss function used: Categorical Cross Entropy

Also called Softmax Loss. It is a Softmax activation plus a Cross-Entropy loss. If we use this loss, we will train a CNN to output a probability over the CC classes for each image/sample. It is used for multi-class classification.



In the specific (and usual) case of Multi-Class classification the labels are one-hot, so only the positive class C_p keeps its term in the loss. There is only one element of the Target vector t which is not zero $t_i = t_p$. So discarding the elements of the summation which are zero due to target labels, we can write:

$$CE = -\log \left(\frac{e^{s_p}}{\sum_j^C e^{s_j}} \right)$$

Where s_p is the CNN score for the positive class.

Defined the loss, now we'll have to compute its gradient respect to the output neurons of the CNN in order to back propagate it through the net and optimize the defined loss function tuning the net parameters. So we need to compute the gradient of CE Loss respect each CNN class score in ss . The loss terms coming from the negative classes are zero. However, the loss gradient respect those negative classes is not cancelled, since the Softmax of the positive class also depends on the negative classes scores.

2.4 Optimization:

A) Stochastic Gradient Descent (SGD):

The word 'stochastic' means a system or a process that is linked with a random probability. Hence, in Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for each iteration. In Gradient Descent, there is a term called "batch" which denotes the total number of samples from a dataset that is used for calculating the gradient for each iteration. In typical Gradient Descent optimization, like Batch Gradient Descent, the batch is taken to be the whole dataset. Although, using the whole dataset is really useful for getting to the minima in a less noisy or less random manner, but the problem arises when our datasets get really huge.

Suppose, you have a million samples in your dataset, so if you use a typical Gradient Descent optimization technique, you will have to use all of the one million samples for completing one iteration while performing the Gradient Descent, and it has to be done for every iteration until the minima is reached. Hence, it becomes computationally very expensive to perform. This problem is solved by Stochastic Gradient Descent. In SGD, it uses only a single sample, i.e., a batch size of one, to perform each iteration. The sample is randomly shuffled and selected for performing the iteration. Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x(i)$ and label $y(i)$:

$$\theta = \theta - \eta \cdot \nabla J_{\theta}(\theta; x(i); y(i)).$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online. So, in SGD, we find out the gradient of the cost function of a single example at each iteration instead of the sum of the gradient of the cost function of all the examples.

In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minima and with significantly shorter training time. One thing to be noted is that, as SGD is generally noisier than typical Gradient Descent, it usually takes a higher number of iterations to reach the minima, because of its randomness in its descent. Even though it requires a higher number of iterations to reach the minima than typical Gradient Descent, it is still computationally much less expensive than typical Gradient Descent. Hence, in most scenarios, SGD is preferred over Batch Gradient Descent for optimizing a learning algorithm.

B) ADAM

The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing. The algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters β_1 and β_2 control the decay rates of these moving averages. The initial value of the moving averages and β_1 and β_2 values close to 1.0 (recommended) result in a bias of moment estimates towards zero. This bias is overcome by first calculating the biased estimates before and then calculating bias-corrected estimates. Adam Configuration Parameters:-

Alpha: Also referred to as the learning rate or step size. The proportion that weights are updated (e.g. 0.001). Larger values (e.g. 0.3) results in faster initial learning before the rate is updated. Smaller values (e.g. 1.0E-5) slow learning right down during training

Beta1: The exponential decay rate for the first moment estimates (e.g. 0.9).

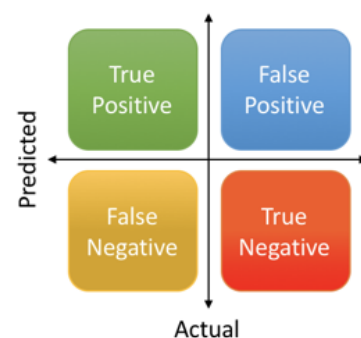
Beta2: The exponential decay rate for the second-moment estimates (e.g. 0.999). This value should be set close to 1.0 on problems with a sparse gradient (e.g. NLP and computer vision problems).

Epsilon: Is a very small number to prevent any division by zero in the implementation (e.g. 10E-8). Further, learning rate decay can also be used with Adam. We may use a decay rate $\alpha = \alpha / \sqrt{t}$ updated each epoch (t) for the logistic regression demonstration.

2.5 Evaluation criteria:

Model Evaluation Metrics:-

$$\begin{aligned}\text{Precision} &= \frac{\text{True Positive}}{\text{Actual Results}} \quad \text{or} \quad \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \\ \text{Recall} &= \frac{\text{True Positive}}{\text{Predicted Results}} \quad \text{or} \quad \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} \\ \text{Accuracy} &= \frac{\text{True Positive} + \text{True Negative}}{\text{Total}}\end{aligned}$$



Often, we think that precision and recall both indicate accuracy of the model. While that is somewhat true, there is a deeper, distinct meaning of each of these terms. Precision means the percentage of your results which are relevant. On the other hand, recall refers to the percentage of total relevant results correctly classified by your algorithm.

To note: Language used: Python.

Platform: Google Colaboratory

To create model: KERAS (Python Deep Learning library).

2.6 Python Code 1 {using Optimizer:- 1) ADAM 2)SGD }

```
[ ] from google.colab import drive
drive.mount('/content/drive/')
```

Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redir

Enter your authorization code:
.....
Mounted at /content/drive/

```
[ ] import os
os.chdir("drive/app")

os.chdir("/content/drive/My Drive/Colab_Notebooks/finaldataset")
```

```
[ ] # This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load in
import math
import random
import pickle
import itertools

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, label_ranking_average_precision_score, label_ranking_loss, coverage_error

from sklearn.utils import shuffle

from scipy.signal import resample

import matplotlib.pyplot as plt

np.random.seed(42)

from sklearn.preprocessing import OneHotEncoder

from keras.models import Model
from keras.layers import Input, Dense, Conv1D, MaxPooling1D, Softmax, Add, Flatten, Activation#, Dropout
from keras import backend as K
from keras.optimizers import Adamax
from keras.callbacks import LearningRateScheduler, ModelCheckpoint

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list the files in the input directory

import os
#print(os.listdir("../input"))
```

```
[ ] df = pd.read_csv('mitbih_train.csv', header=None)
df2 = pd.read_csv('mitbih_test.csv', header=None)
df = pd.concat([df, df2], axis=0)
```

```
[ ] M = df.values
X = M[:, :-1]
y = M[:, -1].astype(int)
print(X.shape)
print(y.shape)
```

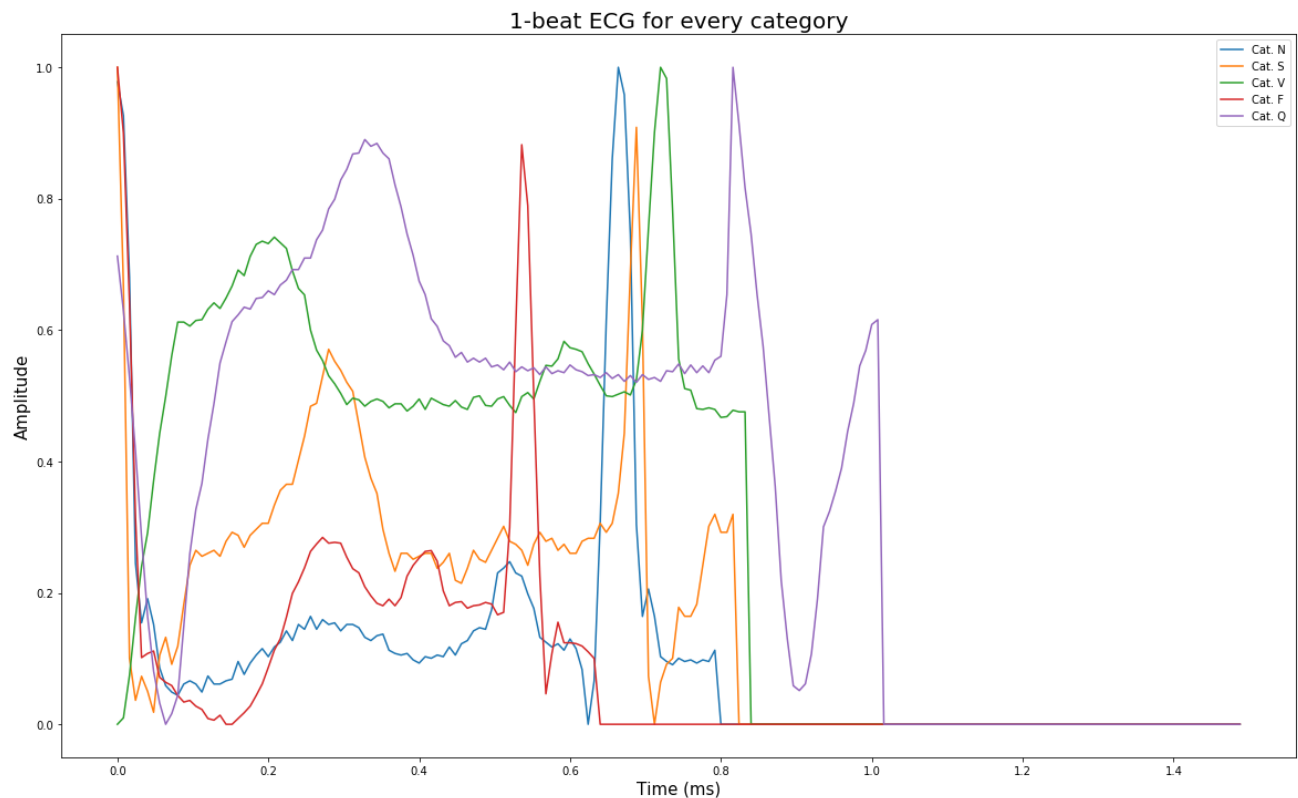
```
(109446, 187)
(109446,)
```

```
[ ] del df
del df2
del M
```

```
[ ] C0 = np.argwhere(y == 0).flatten()
C1 = np.argwhere(y == 1).flatten()
C2 = np.argwhere(y == 2).flatten()
C3 = np.argwhere(y == 3).flatten()
C4 = np.argwhere(y == 4).flatten()
print(C0.shape)
print(C1.shape)
print(C2.shape)
print(C3.shape)
print(C4.shape)
```

```
[ ] x = np.arange(0, 187)*8/1000

plt.figure(figsize=(20,12))
plt.plot(x, X[C0, :][0], label="Cat. N")
plt.plot(x, X[C1, :][0], label="Cat. S")
plt.plot(x, X[C2, :][0], label="Cat. V")
plt.plot(x, X[C3, :][0], label="Cat. F")
plt.plot(x, X[C4, :][0], label="Cat. Q")
plt.legend()
plt.title("1-beat ECG for every category", fontsize=20)
plt.ylabel("Amplitude", fontsize=15)
plt.xlabel("Time (ms)", fontsize=15)
plt.show()
```



```
[ ] def stretch(x):
    l = int(187 * (1 + (random.random()-0.5)/3))
    y = resample(x, l)
    if l < 187:
        y_ = np.zeros(shape=(187, ))
        y_[:l] = y
    else:
        y_ = y[:187]
    return y_

def amplify(x):
    alpha = (random.random()-0.5)
    factor = -alpha*x + (1+alpha)
    return x*factor

def augment(x):
    result = np.zeros(shape= (4, 187))
    for i in range(3):
        if random.random() < 0.33:
            new_y = stretch(x)
        elif random.random() < 0.66:
            new_y = amplify(x)
        else:
            new_y = stretch(x)
            new_y = amplify(new_y)
        result[i, :] = new_y
    return result

plt.plot(X[0, :])
plt.plot(amplify(X[0, :]))
plt.plot(stretch(X[0, :]))
plt.show()
```

```
[ ] ohe = OneHotEncoder(categories='auto')
y_train = ohe.fit_transform(y_train.reshape(-1,1))
y_test = ohe.transform(y_test.reshape(-1,1))

print("X_train", X_train.shape)
print("y_train", y_train.shape)
print("X_test", X_test.shape)
print("y_test", y_test.shape)
```

```
X_train (109150, 187, 1)
y_train (109150, 5)
X_test (4000, 187, 1)
y_test (4000, 5)
```



```
[ ] result = np.apply_along_axis(augment, axis=1, arr=X[C3]).reshape(-1, 187)
    classe = np.ones(shape=(result.shape[0],), dtype=int)*3
    X = np.vstack([X, result])
    y = np.hstack([y, classe])
    print(result.shape)
    print(y.shape)
    print(X.shape)
    #print(result)
```

```
↳ (3212, 187)
   (112658,)
   (112658, 187)
```

```
[ ] subC0 = np.random.choice(C0, 800)
    subC1 = np.random.choice(C1, 800)
    subC2 = np.random.choice(C2, 800)
    subC3 = np.random.choice(C3, 800)
    subC4 = np.random.choice(C4, 800)
    #print(subC0)
```

```
[ ] X_test = np.vstack([X[subC0], X[subC1], X[subC2], X[subC3], X[subC4]])
    y_test = np.hstack([y[subC0], y[subC1], y[subC2], y[subC3], y[subC4]])

    X_train = np.delete(X, [subC0, subC1, subC2, subC3, subC4], axis=0)
    y_train = np.delete(y, [subC0, subC1, subC2, subC3, subC4], axis=0)

    X_train, y_train = shuffle(X_train, y_train, random_state=0)
    X_test, y_test = shuffle(X_test, y_test, random_state=0)

    del X
    del y
    X_train = np.expand_dims(X_train, 2)
    X_test = np.expand_dims(X_test, 2)
    print("X_train", X_train.shape)
    print("y_train", y_train.shape)
    print("X_test", X_test.shape)
    print("y_test", y_test.shape)
```

```
↳ X_train (109150, 187, 1)
   y_train (109150,)
   X_test (4000, 187, 1)
   y_test (4000,)
```

```
[ ] ohe = OneHotEncoder(categories='auto')
    y_train = ohe.fit_transform(y_train.reshape(-1,1))
    y_test = ohe.transform(y_test.reshape(-1,1))

    print("X_train", X_train.shape)
    print("y_train", y_train.shape)
    print("X_test", X_test.shape)
    print("y_test", y_test.shape)
```

```
↳ X_train (109150, 187, 1)
   y_train (109150, 5)
   X_test (4000, 187, 1)
   y_test (4000, 5)
```

```
[ ] n_obs, feature, depth = X_train.shape
    batch_size = 500
```

```

K.clear_session()

inp = Input(shape=(feature, depth))
C = Conv1D(filters=32, kernel_size=5, strides=1)(inp)

C11 = Conv1D(filters=32, kernel_size=5, strides=1, padding='same')(C)
A11 = Activation("relu")(C11)
C12 = Conv1D(filters=32, kernel_size=5, strides=1, padding='same')(A11)
S11 = Add()(C12, C)
A12 = Activation("relu")(S11)
M11 = MaxPooling1D(pool_size=5, strides=2)(A12)

C21 = Conv1D(filters=32, kernel_size=5, strides=1, padding='same')(M11)
A21 = Activation("relu")(C21)
C22 = Conv1D(filters=32, kernel_size=5, strides=1, padding='same')(A21)
S21 = Add()(C22, M11)
A22 = Activation("relu")(S21)
M21 = MaxPooling1D(pool_size=5, strides=2)(A22)

C31 = Conv1D(filters=32, kernel_size=5, strides=1, padding='same')(M21)
A31 = Activation("relu")(C31)
C32 = Conv1D(filters=32, kernel_size=5, strides=1, padding='same')(A31)
S31 = Add()(C32, M21)
A32 = Activation("relu")(S31)
M31 = MaxPooling1D(pool_size=5, strides=2)(A32)

C41 = Conv1D(filters=32, kernel_size=5, strides=1, padding='same')(M31)
A41 = Activation("relu")(C41)
C42 = Conv1D(filters=32, kernel_size=5, strides=1, padding='same')(A41)
S41 = Add()(C42, M31)
A42 = Activation("relu")(S41)
M41 = MaxPooling1D(pool_size=5, strides=2)(A42)

C51 = Conv1D(filters=32, kernel_size=5, strides=1, padding='same')(M41)
A51 = Activation("relu")(C51)
C52 = Conv1D(filters=32, kernel_size=5, strides=1, padding='same')(A51)
S51 = Add()(C52, M41)
A52 = Activation("relu")(S51)
M51 = MaxPooling1D(pool_size=5, strides=2)(A52)

F1 = Flatten()(M51)
D1 = Dense(32)(F1)
A6 = Activation("relu")(D1)
D2 = Dense(32)(A6)
D3 = Dense(5)(D2)
A7 = Softmax()(D3)

model = Model(inputs=inp, outputs=A7)
model.summary()

```

#Using Adam Optimizer

```

[ ] def exp_decay(epoch):
    initial_lrate = 0.001
    k = 0.75
    t = n_obs/(10000 * batch_size) # every epoch we do n_obs/batch_size iteration
    lrate = initial_lrate * math.exp(-k*t)
    return lrate

lrate = LearningRateScheduler(exp_decay)
adam = Adam(lr = 0.001, beta_1 = 0.9, beta_2 = 0.999)

[ ] model.compile(loss='categorical_crossentropy', optimizer=adam, metrics=['accuracy'])

[ ] history = model.fit(X_train, y_train,
                        epochs=75,
                        batch_size=batch_size,
                        verbose=2,
                        validation_data=(X_test, y_test),
                        callbacks=[lrate])

```

Train on 109150 samples, validate on 4000 samples

```
Epoch 1/75
- 8s - loss: 0.3808 - acc: 0.8859 - val_loss: 0.8788 - val_acc: 0.7002
Epoch 2/75
- 7s - loss: 0.1487 - acc: 0.9595 - val_loss: 0.6919 - val_acc: 0.8052
Epoch 3/75
- 7s - loss: 0.1077 - acc: 0.9713 - val_loss: 0.4831 - val_acc: 0.8595
Epoch 4/75
- 7s - loss: 0.0885 - acc: 0.9753 - val_loss: 0.5210 - val_acc: 0.8630
Epoch 5/75
- 7s - loss: 0.0775 - acc: 0.9782 - val_loss: 0.3576 - val_acc: 0.8925
Epoch 6/75
- 7s - loss: 0.0688 - acc: 0.9804 - val_loss: 0.5352 - val_acc: 0.8480
Epoch 7/75
- 7s - loss: 0.0637 - acc: 0.9816 - val_loss: 0.4154 - val_acc: 0.8640
Epoch 8/75
- 7s - loss: 0.0593 - acc: 0.9826 - val_loss: 0.3215 - val_acc: 0.9023
Epoch 9/75
- 7s - loss: 0.0542 - acc: 0.9839 - val_loss: 0.3940 - val_acc: 0.8855
Epoch 10/75
- 7s - loss: 0.0497 - acc: 0.9851 - val_loss: 0.2598 - val_acc: 0.9138
Epoch 11/75
- 7s - loss: 0.0458 - acc: 0.9863 - val_loss: 0.2586 - val_acc: 0.9175
Epoch 12/75
- 7s - loss: 0.0448 - acc: 0.9864 - val_loss: 0.2035 - val_acc: 0.9370
Epoch 13/75
- 7s - loss: 0.0418 - acc: 0.9871 - val_loss: 0.2345 - val_acc: 0.9297
Epoch 14/75
- 7s - loss: 0.0387 - acc: 0.9879 - val_loss: 0.2313 - val_acc: 0.9300
Epoch 15/75
- 7s - loss: 0.0381 - acc: 0.9882 - val_loss: 0.3321 - val_acc: 0.8995
Epoch 16/75
- 7s - loss: 0.0351 - acc: 0.9890 - val_loss: 0.2200 - val_acc: 0.9300
Epoch 17/75
- 7s - loss: 0.0331 - acc: 0.9893 - val_loss: 0.2200 - val_acc: 0.9300
Epoch 18/75
- 7s - loss: 0.0318 - acc: 0.9895 - val_loss: 0.2667 - val_acc: 0.9212
Epoch 19/75
- 7s - loss: 0.0302 - acc: 0.9903 - val_loss: 0.1926 - val_acc: 0.9380
Epoch 20/75
- 7s - loss: 0.0275 - acc: 0.9911 - val_loss: 0.2114 - val_acc: 0.9333
Epoch 21/75
- 7s - loss: 0.0278 - acc: 0.9910 - val_loss: 0.2440 - val_acc: 0.9230
Epoch 22/75
- 7s - loss: 0.0238 - acc: 0.9921 - val_loss: 0.2781 - val_acc: 0.9322
Epoch 23/75
- 7s - loss: 0.0255 - acc: 0.9914 - val_loss: 0.2897 - val_acc: 0.9195
Epoch 24/75
- 7s - loss: 0.0235 - acc: 0.9922 - val_loss: 0.2285 - val_acc: 0.9325
Epoch 25/75
- 7s - loss: 0.0223 - acc: 0.9923 - val_loss: 0.2726 - val_acc: 0.9323
Epoch 26/75
- 7s - loss: 0.0206 - acc: 0.9929 - val_loss: 0.2333 - val_acc: 0.9282
Epoch 27/75
- 7s - loss: 0.0207 - acc: 0.9930 - val_loss: 0.2601 - val_acc: 0.9195
Epoch 28/75
- 7s - loss: 0.0198 - acc: 0.9932 - val_loss: 0.2316 - val_acc: 0.9450
Epoch 29/75
- 7s - loss: 0.0191 - acc: 0.9936 - val_loss: 0.3046 - val_acc: 0.9305
Epoch 30/75
- 7s - loss: 0.0192 - acc: 0.9932 - val_loss: 0.2906 - val_acc: 0.9297
Epoch 31/75
- 7s - loss: 0.0193 - acc: 0.9932 - val_loss: 0.2416 - val_acc: 0.9337
Epoch 32/75
- 7s - loss: 0.0159 - acc: 0.9943 - val_loss: 0.2335 - val_acc: 0.9420
Epoch 33/75
- 7s - loss: 0.0173 - acc: 0.9940 - val_loss: 0.2473 - val_acc: 0.9422
```

Epoch 34/75
- 7s - loss: 0.0165 - acc: 0.9940 - val_loss: 0.2169 - val_acc: 0.9540
Epoch 35/75
- 7s - loss: 0.0151 - acc: 0.9946 - val_loss: 0.2845 - val_acc: 0.9282
Epoch 36/75
- 7s - loss: 0.0171 - acc: 0.9941 - val_loss: 0.2723 - val_acc: 0.9272
Epoch 37/75
- 7s - loss: 0.0148 - acc: 0.9947 - val_loss: 0.2617 - val_acc: 0.9382
Epoch 38/75
- 7s - loss: 0.0142 - acc: 0.9949 - val_loss: 0.2411 - val_acc: 0.9488
Epoch 39/75
- 7s - loss: 0.0140 - acc: 0.9950 - val_loss: 0.3334 - val_acc: 0.9317
Epoch 40/75
- 7s - loss: 0.0141 - acc: 0.9950 - val_loss: 0.2106 - val_acc: 0.9463
Epoch 41/75
- 7s - loss: 0.0126 - acc: 0.9955 - val_loss: 0.2667 - val_acc: 0.9427
Epoch 42/75
- 7s - loss: 0.0157 - acc: 0.9945 - val_loss: 0.3066 - val_acc: 0.9337
Epoch 43/75
- 7s - loss: 0.0113 - acc: 0.9960 - val_loss: 0.2548 - val_acc: 0.9457
Epoch 44/75
- 7s - loss: 0.0118 - acc: 0.9959 - val_loss: 0.2716 - val_acc: 0.9337
Epoch 45/75
- 7s - loss: 0.0125 - acc: 0.9956 - val_loss: 0.2864 - val_acc: 0.9332
Epoch 46/75
- 7s - loss: 0.0118 - acc: 0.9957 - val_loss: 0.2696 - val_acc: 0.9490
Epoch 47/75
- 7s - loss: 0.0127 - acc: 0.9955 - val_loss: 0.2246 - val_acc: 0.9583
Epoch 48/75
- 7s - loss: 0.0116 - acc: 0.9958 - val_loss: 0.3797 - val_acc: 0.9180
Epoch 49/75
- 7s - loss: 0.0116 - acc: 0.9959 - val_loss: 0.3531 - val_acc: 0.9340
Epoch 50/75
- 7s - loss: 0.0115 - acc: 0.9958 - val_loss: 0.2897 - val_acc: 0.9422
Epoch 51/75
- 7s - loss: 0.0101 - acc: 0.9964 - val_loss: 0.2948 - val_acc: 0.9380
Epoch 52/75
- 7s - loss: 0.0103 - acc: 0.9964 - val_loss: 0.2439 - val_acc: 0.9378
Epoch 53/75
- 7s - loss: 0.0106 - acc: 0.9963 - val_loss: 0.2292 - val_acc: 0.9522
Epoch 54/75
- 7s - loss: 0.0104 - acc: 0.9964 - val_loss: 0.2218 - val_acc: 0.9475
Epoch 55/75
- 7s - loss: 0.0124 - acc: 0.9958 - val_loss: 0.3467 - val_acc: 0.9232
Epoch 56/75
- 7s - loss: 0.0104 - acc: 0.9962 - val_loss: 0.2802 - val_acc: 0.9410
Epoch 57/75
- 7s - loss: 0.0088 - acc: 0.9968 - val_loss: 0.3393 - val_acc: 0.9325
Epoch 58/75
- 7s - loss: 0.0095 - acc: 0.9966 - val_loss: 0.3407 - val_acc: 0.9320
Epoch 59/75
- 7s - loss: 0.0103 - acc: 0.9965 - val_loss: 0.2944 - val_acc: 0.9468
Epoch 60/75
- 7s - loss: 0.0108 - acc: 0.9963 - val_loss: 0.3013 - val_acc: 0.9430
Epoch 61/75
- 7s - loss: 0.0095 - acc: 0.9966 - val_loss: 0.3094 - val_acc: 0.9412
Epoch 62/75
- 7s - loss: 0.0099 - acc: 0.9966 - val_loss: 0.3541 - val_acc: 0.9288
Epoch 63/75
- 7s - loss: 0.0122 - acc: 0.9960 - val_loss: 0.3138 - val_acc: 0.9365
Epoch 64/75
- 7s - loss: 0.0081 - acc: 0.9971 - val_loss: 0.3404 - val_acc: 0.9355
Epoch 65/75

Epoch 61/75
- 7s - loss: 0.0095 - acc: 0.9966 - val_loss: 0.3094 - val_acc: 0.9412
Epoch 62/75
- 7s - loss: 0.0099 - acc: 0.9966 - val_loss: 0.3541 - val_acc: 0.9288
Epoch 63/75
- 7s - loss: 0.0122 - acc: 0.9960 - val_loss: 0.3138 - val_acc: 0.9365
Epoch 64/75
- 7s - loss: 0.0081 - acc: 0.9971 - val_loss: 0.3404 - val_acc: 0.9355
Epoch 65/75
- 7s - loss: 0.0078 - acc: 0.9974 - val_loss: 0.2659 - val_acc: 0.9562
Epoch 66/75
- 7s - loss: 0.0073 - acc: 0.9975 - val_loss: 0.4288 - val_acc: 0.9110
Epoch 67/75
- 7s - loss: 0.0111 - acc: 0.9960 - val_loss: 0.2671 - val_acc: 0.9465
Epoch 68/75
- 7s - loss: 0.0097 - acc: 0.9966 - val_loss: 0.3281 - val_acc: 0.9357
Epoch 69/75
- 7s - loss: 0.0084 - acc: 0.9971 - val_loss: 0.2451 - val_acc: 0.9520
Epoch 70/75
- 7s - loss: 0.0072 - acc: 0.9974 - val_loss: 0.3080 - val_acc: 0.9395
Epoch 71/75
- 7s - loss: 0.0100 - acc: 0.9966 - val_loss: 0.2043 - val_acc: 0.9562
Epoch 72/75
- 7s - loss: 0.0085 - acc: 0.9970 - val_loss: 0.3602 - val_acc: 0.9402
Epoch 73/75
- 7s - loss: 0.0078 - acc: 0.9974 - val_loss: 0.3191 - val_acc: 0.9485
Epoch 74/75
- 7s - loss: 0.0091 - acc: 0.9969 - val_loss: 0.3052 - val_acc: 0.9477
Epoch 75/75
- 7s - loss: 0.0086 - acc: 0.9969 - val_loss: 0.3152 - val_acc: 0.9465

```
[26] y_pred=model.predict(X_test, batch_size=1000)
```

```
[27] print(classification_report(y_test.argmax(axis=1), y_pred.argmax(axis=1)))
```

```

           precision    recall  f1-score   support

    0         0.83        1.00        0.91         800
    1         1.00        0.86        0.92         800
    2         0.95        0.96        0.95         800
    3         0.98        0.91        0.95         800
    4         1.00        0.99        0.99         800

 accuracy          0.94        4000
 macro avg         0.95        0.94        0.94        4000
 weighted avg      0.95        0.94        0.94        4000
```

```
[28] print("ranking-based average precision : {:.3f}".format(label_ranking_average_precision_score(y_test.todense(), y_pred)))
print("Ranking loss : {:.3f}".format(label_ranking_loss(y_test.todense(), y_pred)))
print("Coverage_error : {:.3f}".format(coverage_error(y_test.todense(), y_pred)))
```

```

ranking-based average precision : 0.970
Ranking loss : 0.017
Coverage_error : 1.069
```

```
[29] def plot_confusion_matrix(cm, classes,
                             normalize=True,
                             title='Confusion matrix',
                             cmap=plt.cm.Blues):

    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test.argmax(axis=1), y_pred.argmax(axis=1))
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure(figsize=(10, 10))
plot_confusion_matrix(cnf_matrix, classes=['N', 'S', 'V', 'F', 'Q'],
                      title='Confusion matrix, with normalization')

plt.show()
```

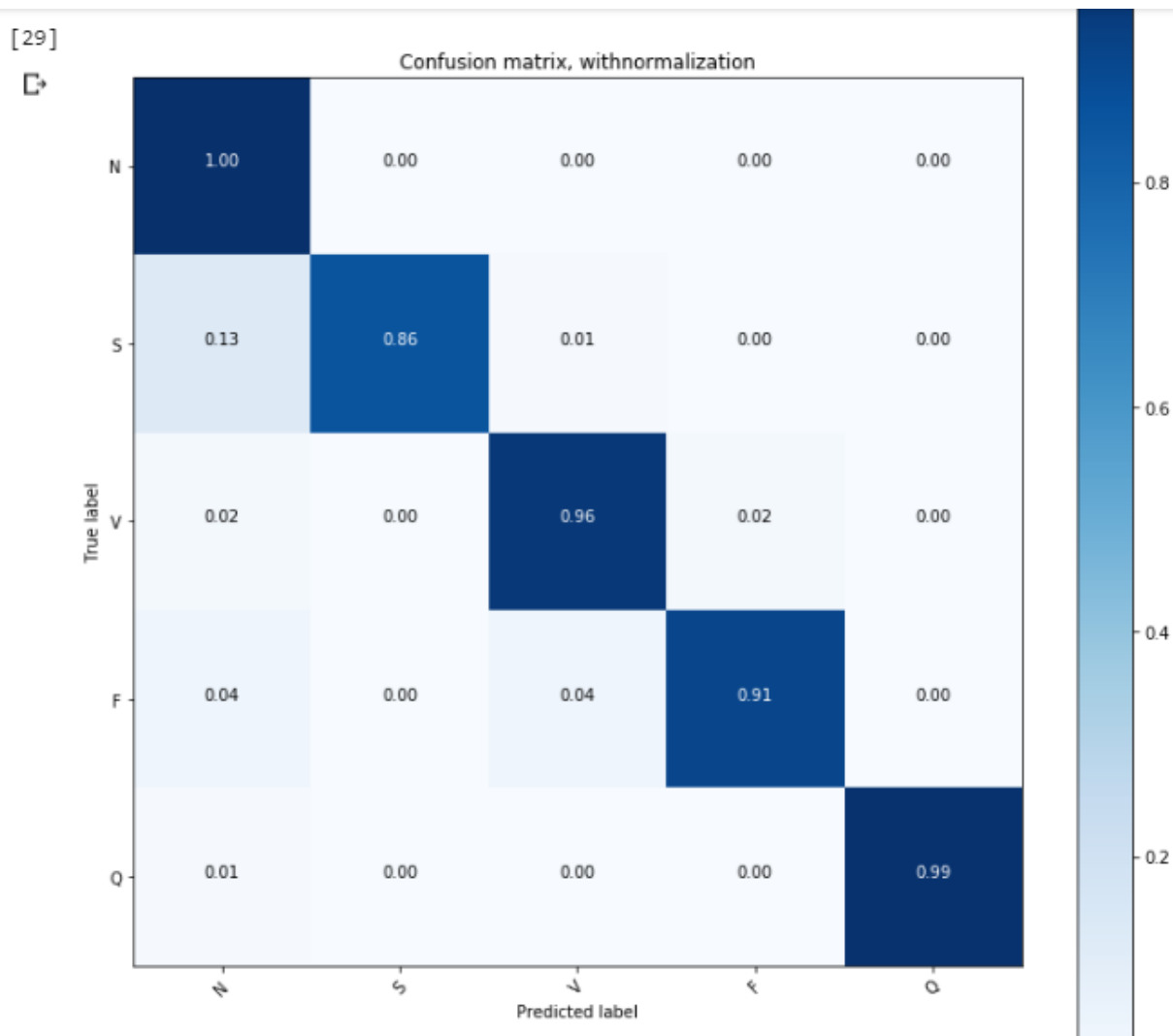


Figure 2.6.1

#Using SGD(Stochiastic gradient descent) Optimizer

```
16] def exp_decay(epoch):
    initial_lrate = 0.001
    k = 0.75
    t = n_obs//((10000 * batch_size) # every epoch we do n_obs/batch_size iteration
    lrate = initial_lrate * math.exp(-k*t)
    return lrate

lrate = LearningRateScheduler(exp_decay)
sgd = SGD(lr=0.001, momentum=0.0, decay=0.0, nesterov=False)

17] model.compile(loss='categorical_crossentropy', optimizer=sgd , metrics=['accuracy'])

18] history = model.fit(X_train, y_train,
                       epochs=75,
                       batch_size=batch_size,
                       verbose=2,
                       validation_data=(X_test, y_test),
                       callbacks=[lrate])
```

```
[19] y_pred=model.predict(X_test, batch_size=1000)
```

```
[20] print(classification_report(y_test.argmax(axis=1), y_pred.argmax(axis=1)))
```

	precision	recall	f1-score	support
0	0.43	0.99	0.60	800
1	0.99	0.33	0.49	800
2	0.85	0.70	0.77	800
3	0.94	0.60	0.74	800
4	0.98	0.90	0.94	800
accuracy			0.70	4000
macro avg	0.84	0.70	0.71	4000
weighted avg	0.84	0.70	0.71	4000

```
[21] print("ranking-based average precision : {:.3f}".format(label_ranking_average_precision_score(y_test.todense(), y_pred)))
print("Ranking loss : {:.3f}".format(label_ranking_loss(y_test.todense(), y_pred)))
print("Coverage_error : {:.3f}".format(coverage_error(y_test.todense(), y_pred)))
```

```
ranking-based average precision : 0.833
Ranking loss : 0.106
Coverage_error : 1.425
```

```
def plot_confusion_matrix(cm, classes,
                          normalize=True,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test.argmax(axis=1), y_pred.argmax(axis=1))
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure(figsize=(10, 10))
plot_confusion_matrix(cnf_matrix, classes=['N', 'S', 'V', 'F', 'Q'],
                      title='Confusion matrix, withnormalization')
plt.show()
```

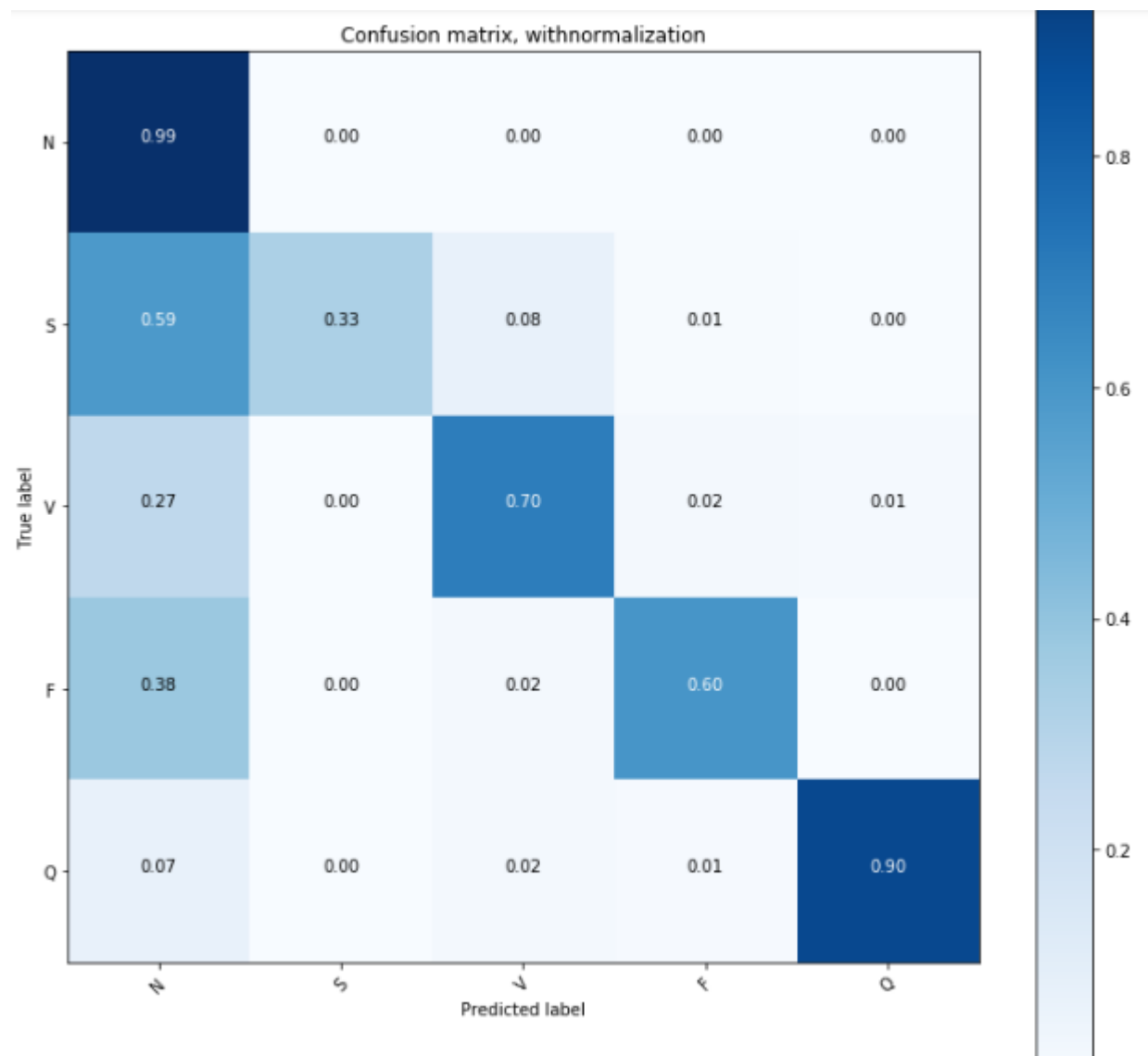



Figure 2.6.2

3. RESULT

- ✚ We evaluated the arrhythmia classifier on 4000 heartbeats (800 from each class) that are not used in the network training phase. Note that the dataset was augmented to reach a balance in the number of beats in each category.
- ✚ Fig. 2.6.1 and Fig. 2.6.2 present the confusion matrix of applying the classifier on the test set. As it can be seen from figures, the model is able to make accurate predictions and distinguish different classes.
- ✚ Average accuracy of 95% (Fig 2.6.1) is achieved with Adam optimizer and the training and of 84% (Fig. 2.6.2) with SGD optimizer.

4. CONCLUSION

- ✚ Deep learning proves to be much accurate in the classification of heartbeats.
- ✚ Accuracy can be further improved with enhanced optimizers. Benefits of ADAM optimization over SGD, as follows:
 1. Straightforward to implement.
 2. Computationally efficient.
 3. Little memory requirements.
 4. Invariant to diagonal rescale of the gradients.
 5. Well suited for problems that are large in terms of data and/or parameters.
 6. Appropriate for non-stationary objectives.
 7. Appropriate for problems with very noisy/or sparse gradients.
 8. Hyper-parameters have intuitive interpretation and typically require little tuning.
- ✚ Considering the above advantages, the choice of ADAM optimization algorithm for deep learning model in place of SGD provides good results in very less time.
- ✚ The classification can also be implemented on other electrical signals or data from sources like image, video etc. and their classification through deep learning algorithms can create intelligent machines aimed to improve human life.

5. Future work that we wish to undertake

- ✚ As concluded, we would like to study optimizers to improve the evaluation metrics of the classification model.
- ✚ We are also motivated to implement CNN or other deep learning architectures:
 - To PPG signals. Using the classification of PPG signals (which can be collected from Pulse sensor) a device can be made to classify the person's heart rate as normal or abnormal, telling him/her the heart risk.
 - To EEG signals to categorize emotions, detect hypertension: studying signals from different sections of brain.

- To build effective systems/model for dyslexic or CP patients to help them learn/adapt to the human behavior/activities.

6. References

1. Kachuee, M. &.-4. (n.d.). ECG Heartbeat Classification: A deep transferable Representation.
2. www.kaggle.com/coni57/model-from-arxiv-1805-00794/notebook; Nicolas Mine, Production Designer at Husky, Thionville, Grand Est, France
3. Keras documentation; <https://keras.io/>
4. Neural Networks, Fuzzy Logic, and Genetic algorithms, Synthesis and application by S.Rajasekaran and G.A. Vijayalakshmi Pai.
5. <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>
6. <http://deeplearning.net/tutorial/lenet.html>
7. <http://cs231n.github.io/convolutional-networks/>
8. adeshpande3.github.io/adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/; Adit Deshpande, UCLA (University of California, Los Angeles)
9. https://github.com/llSourcell/Convolutional_neural_network