**Pre-processing Data**

def  pre\_processing\_data(self, AUGMEN\_NUN):

    for records in self.LABELS:

      with open(records) as record:

        for ecg\_file in tqdm(record):

          path = self.DATA+ecg\_file[:-1]

          metadata = open(path+".hea", "r").read().split(" ")

          ECGs = list(loadmat(path)['val'][0])  
          for i in range(int(self.ECG\_LENGTH+1)):

            ECGs.insert(i, 0)

            ECGs.append(0)

          peaks = detect\_beats(ECGs, float(metadata[2]))

          for peak in range(0, len(peaks),

self.ECG\_PER\_SAMPLE):

            try:

              ECG = ECGs[peaks[peak]-int(self.ECG\_LENGTH/2):

peaks[peak+self.ECG\_PER\_SAMPLE]+int(self.ECG\_LENGTH/2)]

              ECG = self.zero\_padding(self.rnd\_zero(ECG))

              ECG = (ECG + abs(np.amin(ECG)))

              ECG = ECG / np.amax(ECG)

              self.data.append([np.array(ECG),

np.eye(len(self.LABELS))[self.LABELS[records]]])

              for \_ in range(AUGMEN\_NUN):

                aug\_ECG = self.zero\_padding(

self.rnd\_zero(self.resampling(ECG)))

                aug\_ECG = (aug\_ECG + abs(np.amin(aug\_ECG)))

                aug\_ECG = aug\_ECG / np.amax(aug\_ECG)

                self.data.append([np.array(aug\_ECG),

np.eye(len(self.LABELS))[self.LABELS[records]]])

            except Exception as e:

              pass

**Data Augmentation**

def zero\_padding(self, ECG):

    if len(ECG) > self.ECG\_LENGTH:

      return ECG[:self.ECG\_LENGTH]

    for \_ in range(self.ECG\_LENGTH-len(ECG)):

      ECG.append(0)

    return ECG

  def rnd\_bursts(self, ECG):

    for \_ in range(np.random.randint(7)):

      pos = abs(np.random.randint(abs(len(ECG)-11)))

      dist = abs(np.random.randint(7))

      ECG[pos:pos+dist]=[0]\*dist

    return ECG

  def resampling(self, ECG):

    MARGIN = 60

    return signal.resample(ecg,

abs(np.random.randint(MARGIN)+(self.ECG\_LENGTH-MARGIN)))

**Convolutional Neural Network Model**

class Net(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.conv1 = nn.Conv1d(1,180, 5, padding=2)

        self.conv2 = nn.Conv1d(180, 150, 5, padding=2)

        self.conv3 = nn.Conv1d(150, 120, 5, padding=2)

        self.conv4 = nn.Conv1d(120, 90, 5, padding=2)

        self.conv5 = nn.Conv1d(90, 45, 5, padding=2)

        x = torch.randn(1,1,600).view(-1,1,600)

        self.\_to\_linear = None

        self.convs(x)

        self.fc1 = nn.Linear(self.\_to\_linear, 64)

        self.fc2 = nn.Linear(64, 4)

    def convs(self, x):

        x = F.max\_pool1d(F.relu(self.conv1(x)), 3)

        x = F.max\_pool1d(F.relu(self.conv2(x)), 3)

        x = F.max\_pool1d(F.relu(self.conv3(x)), 3)

        x = F.max\_pool1d(F.relu(self.conv4(x)), 3)

        x = F.max\_pool1d(F.relu(self.conv5(x)), 3)

        if self.\_to\_linear is None:

            self.\_to\_linear = x[0].shape[0]\*x[0].shape[1]

        return x

    def forward(self, x):

        x = self.convs(x)

        x = x.view(-1, self.\_to\_linear)

        x = F.relu(self.fc1(x))

        x = self.fc2(x)

        return x