ECG Heartbeat Classification with a Deep Convolutional Neural Network Model

1. Problem Definition

Introduction

The electrocardiogram (ECG) is a standard test used to monitor the activity of the heart. Many cardiac abnormalities will be manifested in the ECG including arrhythmia which is a general term that refers to an abnormal heart rhythm. Heartbeats can be sub-divided into five categories namely non-ectopic (N, **label 0**), supraventricular ectopic (N, **label 0**), ventricular ectopic, fusion, and unknown beats.

Due to this vast difference in morphology, it is difficult to accurately identify ECG components. Furthermore, the visual assessment which is the current standard of care might result in subjective interpretation and inter-observer biases. To address the drawbacks of visual and manual interpretations of ECG, researchers pursue the development of a computer-aided diagnosis (CAD) systems to automatically diagnose ECG. Much of the work in this area has been done by incorporating machine learning approaches for an accurate assessment of ECG and to distinguish lifethreatening from non-threatening events including arrhythmias. In this work, We developed a 12-layer deep convolutional neural network (CNN) to automatically identify 4 different categories of heartbeats in ECG signals.

The ECG Database Given

Upon inspection, we can say that the dataset given is characterized by having highly imbalanced classifications, in which the number of examples in one class greatly outnumbers the examples in another. (E.g. label 0 (Normal) and label 1 (Fusion of ventricular and normal), as shown in the following Figure 1)

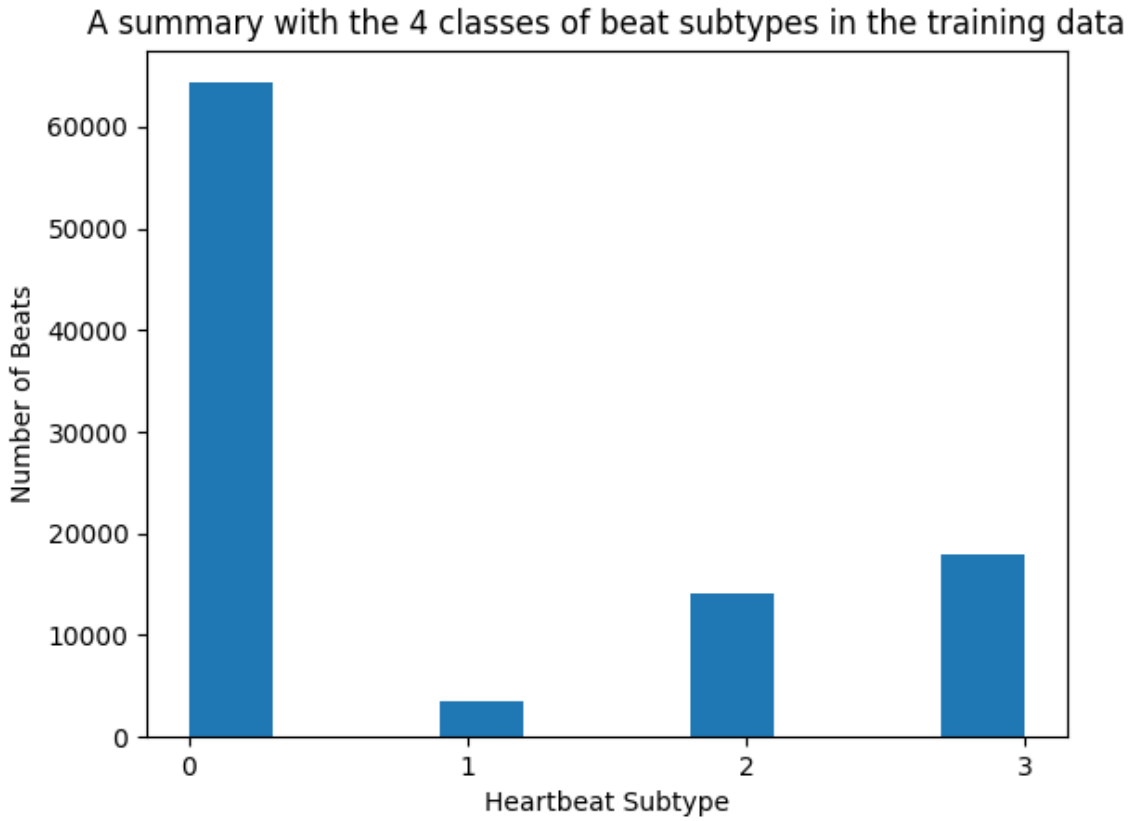

Figure 1

2. Methodology

Pre-processing

The layout of the scripts of our pre-processing related functionalities can be represented as Image 2.

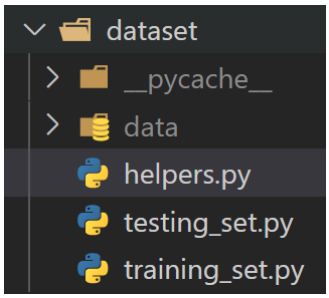

Image 1

The files `testing_set.py` and `training_set.py` defined their respective procedures of input data pre-processing for testing and training datasets, which are specifically merely wrappers of a generic importation function, placed in `helpers.py` and written as below in Code block 1.

```
def import_data(file):
    """create a dataframe and optimize its memory usage"""
    df = pd.read_csv(file)
    fields_length = len(df.columns)
    is_test = fields_length <= 2
    heartbeats_length = 0
    normalized_list = []
    for items in df.values:
```

```
        entry = [items[0]] + [float(i) for i in items[1].split(",")]
        if heartbeats_length == 0:
          heartbeats_length = len(entry) - 1
        elif heartbeats_length != len(entry) - 1:
          raise ValueError(
            "Heartbeats provided are of inconstant length ({}, {})".format(
              heartbeats_length, len(entry) - 1
            )
          )
        normalized_list.append(
          entry if is_test else entry + [items[2]]
        )

    df = reduce_mem_usage(pd.DataFrame(np.array(normalized_list)))
    column_labels = ["id"] + [
        "heartbeat_sample_" + str(i) for i in range(heartbeats_length)
    ]
    df.columns = column_labels if is_test else column_labels + ["label"]
    return df
```

Code block 1

Generation of Synthetic Data

Synthetic data is used to overcome the imbalance in the number of ECG heartbeats in the five (N, S, V, F, Q) classes. Utilizing the SMOTE (synthetic minority oversampling technique) strategy, the samples of synthetic data are generated after preprocessing by varying the standard deviation and mean of Z-score calculated from the original normalized ECG signals.

```
over = SMOTE(sampling_strategy="not majority", n_jobs=-1) # using all processors
steps = [('o', over)]
pipeline = Pipeline(steps=steps)
k_x_train, k_y_train = pipeline.fit_resample(x_train, y_train)
```

Code block 2

Convolutional Neural Network

CNN is one of the most commonly used types of artificial neural networks. Conceptually, a CNN resembles a multilayer perceptron (MLP). Every single neuron in the MLP has an activation function that maps the weighted inputs to the output. An MLP becomes a deep MLP when more than one hidden layer is added to the network. Similarly, the CNN is considered as MLP with a special structure. This special structure allows CNN to be both translation and rotation invariant due to the architecture of the model. There are 3 basic layers– convolutional layer, pooling layer, and fully-connected layer with a rectified linear activation function in a CNN architecture.

Code block 3 summarizes the architecture of the proposed CNN model. There are 12 layers in this network including 6 convolution layers, 2 max-pooling layers, and 4 fully-connected layers.

The `Flatten` function is used to flatten the multi-dimensional input tensors into a single dimension and pass those data into every single neuron of the model effectively.

The `Dropout` layer randomly subsamples the outputs of its previous layer and prevents neural networks from overfitting.

```
class Model(K.Model):
```

```
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = Conv1D(
            filters=16,
            kernel_size=3,
            padding="same",
            activation="relu",
            input_shape=(205, 1),
        )
        self.conv2 = Conv1D(
            filters=32,
            kernel_size=3,
            dilation_rate=2,
            padding="same",
            activation="relu",
        )
        self.conv3 = Conv1D(
            filters=64,
            kernel_size=3,
            dilation_rate=2,
            padding="same",
            activation="relu",
        )
        self.conv4 = Conv1D(
            filters=64,
            kernel_size=5,
            dilation_rate=2,
            padding="same",
            activation="relu",
        )
        self.max_pool1 = MaxPool1D(pool_size=3, strides=2, padding="same")

        self.conv5 = Conv1D(
            filters=128,
            kernel_size=5,
            dilation_rate=2,
            padding="same",
            activation="relu",
        )
        self.conv6 = Conv1D(
            filters=128,
            kernel_size=5,
            dilation_rate=2,
            padding="same",
            activation="relu",
        )
        self.max_pool2 = MaxPool1D(pool_size=3, strides=2, padding="same")

        # Dropout works by probabilistically removing,
        # or “dropping out,” inputs to a layer, which
        # may be input variables in the data sample or
        # activations from a previous layer. It has the
        # effect of simulating a large number of networks
        # with a very different network structure and,
        # in turn, making nodes in the network generally
        # more robust to the inputs.
        self.dropout = Dropout(.5)
        self.flatten = Flatten()
```

```
        self.fc1 = Dense(units=256, activation="relu")
        self.fc21 = Dense(units=16, activation="relu")
        self.fc22 = Dense(units=256, activation="sigmoid")
        self.fc3 = Dense(units=4, activation="softmax")

    def call(self, inputs, training=None):
        inputs = self.conv1(inputs)
        inputs = self.conv2(inputs)
        inputs = self.conv3(inputs)
        inputs = self.conv4(inputs)
        inputs = self.max_pool1(inputs)

        inputs = self.conv5(inputs)
        inputs = self.conv6(inputs)
        inputs = self.max_pool2(inputs)

        # Some layers, in particular the BatchNormalization
        # layer and the Dropout layer, have different
        # behaviors during training and inference. For such
        # layers, it is standard practice to expose a training
        # (boolean) argument in the call() method.
        # https://www.tensorflow.org/guide/keras/custom_layers_and_models
        if training:
          inputs = self.dropout(inputs)
        inputs = self.flatten(inputs)

        x1 = self.fc1(inputs)
        x2 = self.fc22(self.fc21(inputs))
        inputs = self.fc3(x1 + x2)

        return inputs
```

Code block 3

3. Results and Discussion

Table 1 presents the result of our best run.

epoch	loss	sparse_categorical_accuracy	lr
1	0.10173320025205612	0.9639303684234619	0.001
2	0.028207074850797653	0.9909369349479675	0.0009
3	0.017938371747732162	0.9942442774772644	0.00081
4	0.012583727948367596	0.9958998560905457	0.000729
5	0.009770846925675869	0.9969413876533508	0.0006561
6	0.00722897332161665	0.997687578201294	0.00059049
7	0.005849012639373541	0.9982900023460388	0.000531441
8	0.005150227341800928	0.9983949065208435	0.0004782969
9	0.0036295480094850063	0.9989156723022461	0.00043046722
10	0.002966266358271241	0.9991216659545898	0.0003874205
11	0.0028137932531535625	0.9990944862365723	0.00034867844
12	0.0019505321979522705	0.9994131326675415	0.0003138106
13	0.0018737009959295392	0.9994170665740967	0.00028242954
14	0.001579349976964295	0.9994986653327942	0.00025418657
15	0.001210228307172656	0.9996696710586548	0.00022876792
16	0.0012479174183681607	0.9996618628501892	0.00020589113
17	0.0008561256690882146	0.9996891021728516	0.00018530202
18	0.0010170132154598832	0.999692976474762	0.00016677182
19	0.0007644527358934283	0.9998056888580322	0.00015009464
20	0.0006180760683491826	0.999813437461853	0.00013508517
21	0.0006319472449831665	0.9998367428779602	0.00012157665
22	0.0005338459741324186	0.999825119972229	0.00010941899
23	0.00037391355726867914	0.9999028444290161	0.00009847709
24	0.0002730062697082758	0.9999028444290161	0.00008862938
25	0.00030466532916761935	0.9999067187309265	0.00007976644
26	0.00041382311610504985	0.9998911619186401	0.0000717898
27	0.0002459393872413784	0.9999105930328369	0.00006461082
28	0.00022377331333700567	0.9999184012413025	0.000058149737
29	0.0001831453846534714	0.9999261498451233	0.000052334763
30	0.00025139781064353883	0.9999184012413025	0.000047101286

Table 1
Optimizer: Adam
Hyperparameters: epoch 30 and batch size 64
Learning rate scheduler formula: lr = 0.001 * math.exp(-0.1 * epoch)
Output of command "time": real 33m1.562s, user 29m38.343s, sys 9m33.080s
Score: 174. Percentage of Correct Predictions on the testing dataset: 99.13%

With such irregularly high accuracies in our training, it is evident that our model has a substantial tendency for overfitting regarding the features specified and our redeeming strategy of suppling a relative small initial learning rate and a vastly learning-rate-dropping scheduler is necessary.

Also, skewed (imbalanced) data sets can negatively affect the overall performance of conventional and CNN-based classification systems. In order to alleviate this issue, synthetic data was generated to ensure that the number of samples in each class is proportional. It is worth mentioning that the real-world effectiveness of our current data augmentation method (SMOTE) is yet to be assessed.

4. Future Work

Many different adaptations, tests, and experiments have been left for the future due to lack of time in this work, namely the data augmentation methods and the CNN models. Additionally, untested methods like model stacking could also improve the problematic high variance and little bias of our model. As so, the following ideas could be tested:

- SMOTE Variants such as SMOTE-NC, Borderline-SMOTE, SVM-SMOTE and ADASYN[1]
- Learned or specialized data augmentation policies
- CNN models with a different implementation. Our assumption is that a shallower CNN model could help with our overfitting problem.
- Ensembling multiple models

5. Bilbiography

A Deep Convolutional Neural Network Model to Classify Heartbeats.

1. SMOTE Variants for Imbalanced Binary Classification: Heart Disease Prediction