Predict Heart Disease with Machine Learning Methods

Zixuan Zhong
SUSTech
ShenZhen, China
12112707@mail.sustech.edu.cn

***Abstract*—More and more people are concerned about their own heart condition. With the popularity of smart health detection devices such as sports handrings, it is necessary to improve the prediction of user's heart state. In this project, we used Logistic Regression, CNN, LSTM, and attention+LSTM four models to predict heart status from ECG signals. The experiment results show that the best results come from CNN and Logistic Regression. LSTM itself is not good enough, but it becomes better with self attention. However, it is still not as good as the former two.**

***Index Terms*—ECG signal, machine learning, classfication, predction**

I. Introduction

With the increasing emphasis on health management, health testing devices are moving from the laboratory to the consumer market. More and more wearable mobile health devices appear in the consumers'view, such as health bracelets, home blood pressure meters, and so on. ECG is a biological monitoring technology applied by the mainstream health management products on the market.

Using machine learning technology, we can create models to predict the heart state of a person to be tested and thereby prevent heart disease. In this project, we learned 188 features about the heart and classified the state of a heart by these 188 features. 'Normal','Atrial Fibrillation','Non-AF related abnormal heart rhythms', and 'noise recording' are the four states that we need to predict. We already have a sample ECG signal set, which has 8528 samples. The first 188 colums are input features and the last column of the sample is 1, 2, 3, 4 for each of the four target features.

In this project, we use pytorch to build four models, feed forward deep network, CNN network, LSTM network and attention + LSTM network. Then, the algorithm is validated by 5-fold cross validation, that is, taking turns catch 4/5 of the sample set as input set, and the rest 1/5 of the sample set as validation set. Through this 5-fold cross validation to test the accuracy of each model.

II. Problem Formulation

A. Metrics

We use three metrics to show whether the algorithm is good or bad. They are Recall, Precision, and F1.

- TP(true positive): positive examples classified as positive
- FP(false positive): negative examples classified as positive
- FN(false negative): positive examples classified as negative
- TN(true negative): negative examples classified as negative

$$Recall = TP/(TP + FN) \tag{1}$$

$$Precision = TP/(TP + FP) \tag{2}$$

$$F1 = 2 * Precision * Recall/(Precision + Recall) \tag{3}$$

In this projectwe define F1-score of normal as Fn, F1 of Atrial Fibrillation as Fa, F1 of other rhythms as Fo. We use the mean of these three called FT

$$FT = 1/3(Fn + Fa + Fo) \tag{4}$$

as the accuracy.Therefore, we consider the larger of FT, the better of this model's performance.

B. unbalanced dataset

This project gives us a largely imbalanced dataset, so we need to adjust the weight to increase the training effect of a small amount of data. Because less data is important to consider for the model, the formula for defining weight is number of current data in the set/ largest number of data in the set.

Considering samples which targets are 1, 2, 3, 4 have the number n1, n2, n3, n4. Thus define the weight of each sample is

$$n(max) = max(n1, n2, n3, n4) \tag{5}$$

$$n_i = n(max)/n_i \tag{6}$$

C. CrossEntropyLoss

we use CrossEntropyLoss as loss function in four models.

$$H(p, q) = -\sum_{i=1}^{n} p(x_i) log q(x_i) \tag{7}$$

D. Softmax function

When calculating the target corresponding to sample, we need to use Softmax function to normalize the results.

$$Softmax(z_i) = \frac{exp(z_i)}{\sum_{n=1}^{m} exp(z_n)} \tag{8}$$

III. Method and Algorithms

A. File IO

Since the data is saved in the. csv file, I use panda to import the data. Read the first 188 rows of data as input data, read the last row as targets, and map 1234 to 0123 for subsequent processing.

Listing 1: Read csv file

```
df = pd.read_csv('FinalProject/data.csv',
    header = None)
inputs = torch.from_numpy(np.array(df)
    [:,0:188])
targets = torch.from_numpy(np.array(df)
    [:,188].astype(np.int64)) - 1
```

B. Data pre-process

After reading the data, we divide the data into five equal copies, each of which contains 1705 data.We take four copies each time as the training set and one as the validation set. Five GetData and TestSet dataset are created to manage the trainningset and validation set. Each dataset is inherited from the dataset in torch.utils.data, and has an initialization method. I independently define method getWeight(). It returns the weight of each data in each training set.

Listing 2: get weight

```
def getWeight(self):
    maximum = max(count(self.label,0), count(
        self.label,1), count(self.label,2),
        count(self.label,3))
    return torch.from_numpy(np.array([maximum/
        count(self.label,0),maximum/count(self
        .label,1),
    maximum/count(self.label,2),maximum/count(
        self.label,3)])).float()

def count(y,x):
    y = y.flatten()
    c = 0
    for i in y:
        if i == x:
            c = c + 1
    return c
```

C. Definitions of Mecrics

define the F1 by Python

Listing 3: Recall

```
def Recall(x,y,correct):
    x = x.flatten()
    xsum = 0
    ysum = 0
    j = 0
    for i in y:
        if i == correct:
            ysum = ysum + 1
            if x[j] == correct:
                xsum = xsum + 1
        j = j + 1
    return float(xsum/ysum)
```

Listing 4: Precision

```
def Precision(x,y,correct):
    x = x.flatten()
    xsum = 0
    ysum = 0
    j = 0
    for i in y:
        if i == correct:
            ysum = ysum + 1
            if x[j] == correct:
                xsum = xsum + 1
        j = j + 1
    return float(xsum/ysum)
```

Listing 5: F1

```
def F1(x,y,correct):
    r = Recall(x,y,correct)
    p = Precision(x,y,correct)
    return 2*p*r/(p+r)
```

D. Deep neural network

Due to my previous experience, I decided to complete the model building and training of logistic regression.

The principle of network can be expressed by the following formula

$$z = wx + b \tag{9}$$

$$y = f(z) \tag{10}$$

Common f are Relu and Sigmoid ...

$$Relu(x) = \frac{|x| + x}{2} \tag{11}$$

$$Sigmoid(x) = \frac{1}{1 + exp(-x)} \tag{12}$$

In this project, we basically use Relu as activation function. Here is the model of Logistic Regression model:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(188, 100)
        self.fc2 = nn.Linear(100, 4)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

- First, inherit the nn.Module class and establish a two-layer linear network. Set the mapping relationship as 188 to 100 for the first layer and 100 to 4 for the second layer, and use pythoch.relu as the activation function, so that nonlinear transfer can be realized in the part less than 0.
- Determine optimizer. In this project, I used Adam (Adaptive Moment Estimation) as the optimizer. Its advantage is that after offset correction, the learning rate of each iteration has a certain range, making the parameters relatively stable.
- Determine criterions. When calculating loss, I used nn. CrossEntropyLoss. MSELoss was tried in the previous classification model, but the result was not very good. After searching the relevant information, it is found that MESLoss has a lower tolerance for large residuals, which leads to the noise data in the sample will have a greater impact on the results, resulting in unsatisfactory training results. Therefore, CrossEntropyLoss is more suitable for this multi classification situation.
- Since the targets data (0/1/2/3) cannot be directly involved in the calculation, I converted the targets to one-hotted before calculating. For example, 0 is converted to [1 0 0 0], 1 is converted to [0 1 0 0], and so on. The corresponding conversion code is as follows:

Listing 6: One-hotted Encoding format

```
target2 = torch.zeros(batch_size, 4).
    scatter_(1, target.unsqueeze(1), 1)
```

E. Convolutional Neural Network

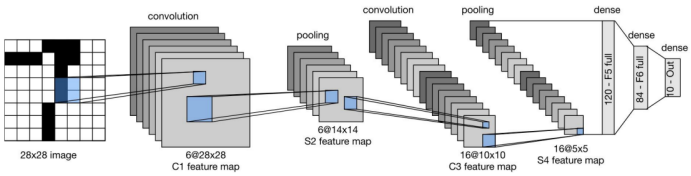

Fig. 1: principle picture of CNN

Create CNNclass, inherit nn. Module(), get 16 kernels after two convolutions, and set the kernel size=3, MaxPool1d size=3, and then output the result through a two-layer neural network. Activation function is still Relu. The code is as follows:

Listing 7: Convolutional Neural Network

```
class cnn(nn.Module):
    def __init__(self):
        super(cnn, self).__init__()
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
        self.conv1 = nn.Sequential(nn.Conv1d(
            in_channels=1,out_channels=6,
            kernel_size=3),nn.ReLU(),nn.
            MaxPool1d(kernel_size=2))
        self.conv2 = nn.Sequential(nn.Conv1d(
            in_channels=6,out_channels=16,
            kernel_size=3),nn.ReLU(),nn.
            MaxPool1d(kernel_size=2))
        self.fc1 = nn.Linear(720, 100)
        self.fc2 = nn.Linear(100,4)

    def forward(self, x):
        x = x.reshape(batch_size, 1, 188)
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

F. Recurrent Neural Networks

Compared with other network models, RNN model has the concept of ”time”: The input order of data will affect the results. So RNN is often used for voice recognition, translation and other functions.

Common RNN models include LSTM; GRU, Grid LSTM, etc. We uses LSTM to build the model in this project.

```
class LstmRNN(nn.Module):
    def __init__(self, input_size, hidden_size
        =1, output_size=1, num_layers=1):
        super().__init__()
        self.lstm = nn.LSTM(input_size,
            hidden_size, num_layers) # utilize
             the LSTM model in torch.nn
        self.fc1 = nn.Linear(hidden_size,10)
        self.forwardCalculation = nn.Linear
            (10, output_size)

    def forward(self, _x):
        x, _ = self.lstm(_x)
        x = F.relu(self.fc1(x))
        x = self.forwardCalculation(x)
        return x
```

G. Self-Attention

Self-Attention is actually a series of attention distribution coefficients.Its function is to judge the relationship between the input data, so as to give greater weight to the data with greater impact on the results

- First, define SelfAttention () in the Attention file, and then determine to initialize Q, K, and V matrices. Then determine that the attention scoring function is dot product, multiply Q and K matrices, and then multiply the result with the corresponding V matrix to obtain the attention coefficient of the current data.

```
class SelfAttention(nn.Module):
    def __init__(self, num_attention_heads,
        input_size, hidden_size,
        hidden_dropout_prob):
        super(SelfAttention, self).__init__()
        self.num_attention_heads =
            num_attention_heads
        self.attention_head_size = int(
            hidden_size / num_attention_heads)
```

```
        self.all_head_size = hidden_size
        self.query = nn.Linear(input_size,
            self.all_head_size)
        self.key = nn.Linear(input_size, self.
            all_head_size)
        self.value = nn.Linear(input_size,
            self.all_head_size)
        self.attn_dropout = nn.Dropout(p=0.3)
        self.dense = nn.Linear(hidden_size,
            hidden_size)
        self.LayerNorm = LayerNorm(hidden_size
            , eps=1e-12) #
              L a y e r N o r m loss
        self.out_dropout = nn.Dropout(
            hidden_dropout_prob)

    def transpose_for_scores(self, x):
        new_x_shape = x.size()[:-1] + (self.
            num_attention_heads, self.
            attention_head_size)
        x = x.view(*new_x_shape)
        return x.permute(0, 2, 1)

    def forward(self, input_tensor):
        mixed_query_layer = self.query(
            input_tensor)
        mixed_key_layer = self.key(
            input_tensor)
        mixed_value_layer = self.value(
            input_tensor)
        query_layer = self.
            transpose_for_scores(
            mixed_query_layer)
        key_layer = self.transpose_for_scores(
            mixed_key_layer)
        value_layer = self.
            transpose_for_scores(
            mixed_value_layer)
        attention_scores = torch.matmul(
            query_layer, key_layer.transpose
            (-1, -2))
        attention_scores = attention_scores /
            math.sqrt(self.attention_head_size
            )
        attention_probs = nn.Softmax(dim=-1)(
            attention_scores)
        attention_probs = self.attn_dropout(
            attention_probs)
        context_layer = torch.matmul(
            attention_probs, value_layer)
        context_layer = context_layer.permute
            (0, 2, 1).contiguous()  #(0,2,1,3)
        new_context_layer_shape =
            context_layer.size()[:-2] + (self.
            all_head_size,)
        context_layer = context_layer.view(*
            new_context_layer_shape)
        hidden_states = self.dense(
            context_layer)
        hidden_states = self.out_dropout(
            hidden_states)
        hidden_states = self.LayerNorm(
            hidden_states + input_tensor)

        return hidden_states
```

- Then, apply self-attention to improve LSTM method. Import the SelfAttention class into LSTM, and firstly calculate the attention weight of the input data. The remaining LSTM parts remain unchanged.

```
data = attention(data)
```

IV. Experiment Results and Analysis

A. Deep neural network

Here is the best F1-score and loss of Deep neural network. Hyperparameter:

- batch-size = 20
- learning-rate = 1e-4
- epochs = 100
- seed = 1000
- optimiser = Adam(with weight)
- criterion = CrossEntropyLoss

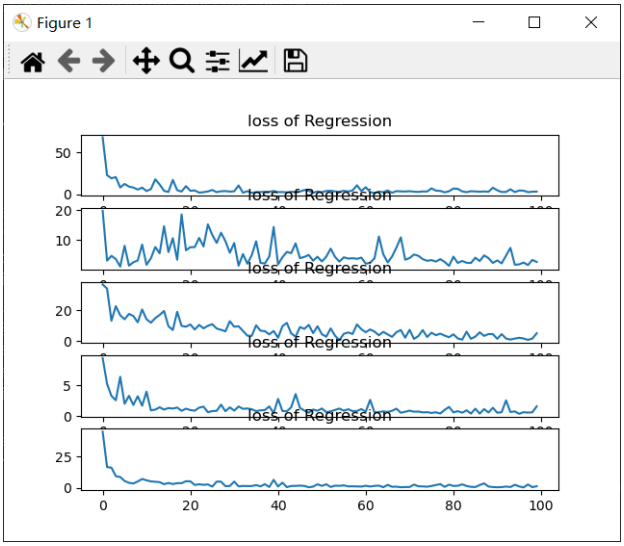

Fig. 2: Loss of deep neural network.

Except that the loss of the second dataset is not very stable, other losses are basically convergent.

TABLE I: F1 of Logistic Regression

	F_n	F_a	F_o	F_T
Set1	0.91	0.90	0.61	0.80
Set2	0.94	0.84	0.50	0.76
Set3	0.87	0.66	0.71	0.74
Set4	0.91	0.83	0.63	0.79
Set5	0.89	0.81	0.65	0.78

From the results, the prediction accuracy of the model is up to 0.8. Among them, Set2 and Set3 with poor loss decrease are obviously in Fo and Fa's learning result, and finally resulting in low prediction accuracy of FT.

B. Convolutional neural network

Here is the best F1-score and loss of Deep neural network.
Hyperparameter:

- batch-size = 20
- learning-rate = 3e-5
- epochs = 80
- seed = 1000
- optimiser = Adam(with weight)
- criterion = CrossEntropyLoss

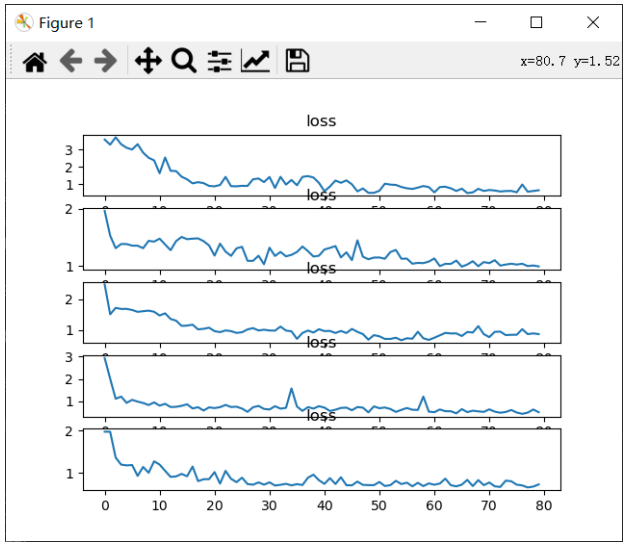

Fig. 3: Loss of CNN.

Observe the loss image. It is found that the gradient will rise briefly when it drops. At first I thought it was because learning rate is large. But it still occurs even if I enlarge decrease the learning-rate.So it is confirmed that this is due to the impact of the large weight sample on training process.

TABLE II: F1 of CNN

	F_n	F_a	F_o	F_T
Set1	0.89	0.77	0.72	0.79
Set2	0.90	0.78	0.71	0.80
Set3	0.88	0.87	0.74	0.83
Set4	0.93	0.79	0.62	0.78
Set5	0.92	0.85	0.60	0.79

The training result of CNN method is good, the average accuracy can reach 0.8, and the best sample set accuracy can reach 0.83.

C. Recurrent Neural Networks

Here is the best F1-score and loss of LSTM.
Hyperparameter:

- batch-size = 20
- learning-rate = 1e-4
- epochs = 50
- seed = 1000
- optimiser = Adam(with weight)
- criterion = CrossEntropyLoss
- hidden-size = 16

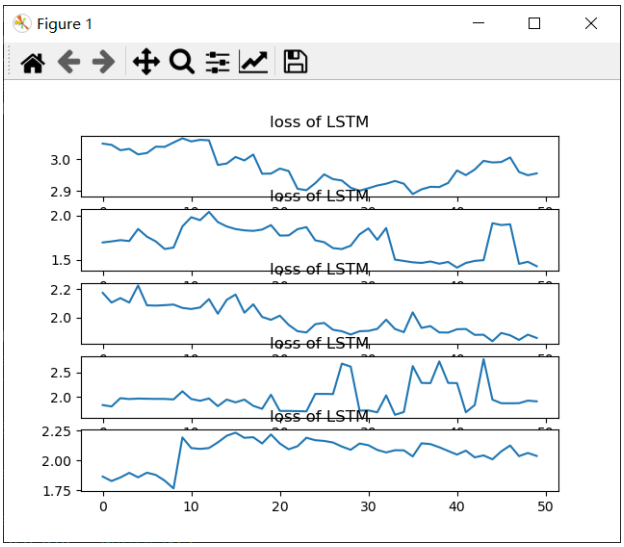

Fig. 4: Loss of LSTM.

TABLE III: F1 of LSTM

	F_n	F_a	F_o	F_T
Set1	0.78	0.25	0.72	0.42
Set2	0.82	0.25	0.47	0.51
Set3	0.80	0.29	0.44	0.51
Set4	0.82	0.31	0.46	0.53
Set5	0.82	0.34	0.48	0.54

According to the analysis of the non convergence of loss and the poor situation where the accuracy of the model is about 0.5, LSTM is not suitable for this situation. Because the RNN model itself has a certain concept of 'time', the model is more suitable for data sets whose output results are related to order. In this project, all input data are independent of each other and have no strong connection, so LSTM prediction accuracy is poor and loss is not convergent.

D. Recurrent Neural Networks and Attention

Here is the best F1-score and loss of LSTM+Attention.
Hyperparameter:

- batch-size = 20
- learning-rate = 1e-3
- epochs = 60
- seed = 1000
- optimiser = Adam(with weight)
- criterion = CrossEntropyLoss
- hidden-size = 16

Compared with LSTM, the loss of LSTM with Attention decreases more smoothly, and the output accuracy increases by nearly 10 percents, from 50 percents to 60 percents.

That's because: By selfattention, the network can pay more attention to the important parameters in the input set, so that

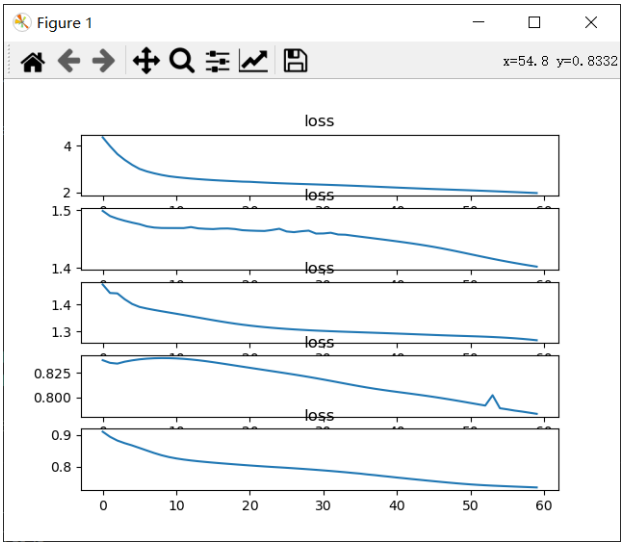

Fig. 5: Loss of LSTM+Attention.

TABLE IV: F1 of LSTM+Attention

	F_n	F_a	F_o	F_T
Set1	0.76	0.51	0.65	0.64
Set2	0.74	0.48	0.67	0.63
Set3	0.75	0.34	0.60	0.56
Set4	0.74	0.41	0.61	0.59
Set5	0.75	0.43	0.61	0.60

the prediction accuracy of the model is higher and the accuracy is improved. And layerNorm is added through attention to make the loss more stable.

E. Comparision of four methods

Here, I make a table to compare the performance of four methods through mean F1, maximum F1 and convergence:

TABLE V: Comparison of four models

	Mean	Max	Convergence
FNN	0.78	0.81	convergent
CNN	0.80	0.83	convergent
RNN	0.52	0.54	divergent
Att+RNN	0.60	0.64	divergent

By comparing the above four models, CNN and FNN are ideal solutions for this classification problem. The processing effect of CNN is slightly better than that of FNN, but its computing speed is far slower than that of FNN. For large amounts of data, FNN may have be more pratical. RNN is not used to deal with this non sequential problem. Therefore, even if self Attention is added to improve RNN, the is still inferior to the previous two methods. Moreover, RNN has the slowest calculation speed due to the large relationship between data.

V. Conclusion and Future Problems

A. Conclusion

For data sets with small data volume, we use CNN method to accurately predict the results. For large data sets, we use FNN to give up a small amount of precision in exchange for fast model solution. The RNN model is not applicable to this project.

B. Future problems

1) Better Model: Due to the noise and data imbalance, the accuracy of prediction is difficult to be higher than 0.8. There may be a better model for data fitting to make the accuracy higher.

2) Hyperparameters optimization: Because of deadline is coming, I have no more time to adjust to the best parameters. And because of limit of poor computer, I set a bit of large learning rate to reduce the training epoches to save time. Adjusting the lower learning-rate can improve the performance of these models.

3) More Training Data: The dataset can be larger. We can put more information of samples in the dataset to get better performance even though they seems no connection with heart. And if we can add a real physics model with correct connection, the accuracy of prediction could be better.

Acknowledgment

The work is supported by Google, Zhihu and CSDN.
Some figures come from Lin's in class PPT.

References

[1] Marzog Heyam A.; Abd Haider. J. Machine Learning ECG Classification Using Wavelet Scattering of Feature Extraction, 10.1155/2022/9884076, pp. 62-65

[2] Marzog Heyam A.; Abd Haider. J.,Simple and Accurate Dependency Parsing Using Bidirectional LSTM Feature Representations, Electrical Engineering Department,College of Engineering,University of Babylon,Hilla,Babil,Iraq; Engineering Technical College/Najaf,Al-Furat Al-Awsat Technical University,Al Najaf 31001,Iraq,10.1155/2022/9884076,pp261-265