In [40]:
```
import os
import pickle
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import TensorDataset, Dataset
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

import warnings
warnings.filterwarnings("ignore")

PATH_DATA = "./data/mortality/data/"
PATH_VALIDATION = "./data/mortality/validation/"
PATH_OUTPUT = "./output/"
```

Mortality Prediction Based on RNN with GRU units

1. Data Overview

- Data source: MIMIC-III https://mimic.physionet.org/ (https://mimic.physionet.org/)

In [41]:
```
df_mortality = pd.read_csv(os.path.join(PATH_DATA, "MORTALITY.csv"))
df_admissions = pd.read_csv(os.path.join(PATH_DATA, "ADMISSIONS.csv"))
df_diagnoses = pd.read_csv(os.path.join(PATH_DATA, "DIAGNOSES_ICD.csv"))
```

In [42]:
```
df_mortality.head()
```

Out[42]:

	SUBJECT_ID	MORTALITY
0	252	1
1	721	1
2	776	1
3	801	0
4	822	1

In [43]:
```
df_admissions.head()
```

Out[43]:

	ROW_ID	SUBJECT_ID	HADM_ID	ADMITTIME	DISCHTIME	DEATHTIME	ADMISSION_TYPE	ADI
0	22	23	152223	2153-09-03 07:15:00	2153-09-08 19:10:00	NaN	ELECTIVE	
1	23	23	124321	2157-10-18 19:34:00	2157-10-25 14:00:00	NaN	EMERGENCY	
2	33	34	115799	2186-07-18 16:46:00	2186-07-20 16:00:00	NaN	EMERGENCY	
3	34	34	144319	2191-02-23 05:23:00	2191-02-25 20:20:00	NaN	EMERGENCY	REF
4	36	36	182104	2131-04-30 07:15:00	2131-05-08 14:00:00	NaN	EMERGENCY	REF

In [44]:
```
df_diagnoses.head()
```

Out[44]:

	ROW_ID	SUBJECT_ID	HADM_ID	SEQ_NUM	ICD9_CODE
0	1523	117	140784	1.0	5715
1	1524	117	140784	2.0	7895
2	1525	117	140784	3.0	07054
3	1526	117	140784	4.0	2875
4	1527	117	140784	5.0	4280

2. Data preprocessing

2.1 Build original dataset

- `extract_code`
  - follow ICD-9-CM format (https://www.cms.gov/Medicare/Quality-Initiatives-Patient-Assessment-Instruments/HospitalQualityInits/Downloads/HospitalAppendix_F.pdf (https://www.cms.gov/Medicare/Quality-Initiatives-Patient-Assessment-Instruments/HospitalQualityInits/Downloads/HospitalAppendix_F.pdf)) to extract main digits of ICD-9 code
- `build_dict`
  - create corresponding id map for ICD-9 code, {main digits of ICD9: unique feature ID}
- `build_dataset`
  - group the diagnosis codes for the same visit
  - group the visits for same patient
  - make visit lists for paitents with chronological order

- return: List(patient IDs), List(labels), Visit sequence data as a List of List of List.

In [45]:
```
def extract_code(icd9_object):

    icd9_str = str(icd9_object)
    if icd9_str[0] == 'E': icd9_str = icd9_str[0:4]
    else: icd9_str = icd9_str[0:3]

    return icd9_str
```

In [46]:
```
def build_dict(df_icd9, extract_func):

    unique_code = df_icd9['ICD9_CODE'].apply(extract_func).unique()
    unique_code = pd.Series(unique_code).sort_values()
    code_dict = dict(zip(unique_code, range(len(unique_code))))

    return code_dict
```

In [47]:
```
def build_dataset(path, code_dict, extract_func):

    df_mortality = pd.read_csv(os.path.join(path, "MORTALITY.csv"))
    df_admissions = pd.read_csv(os.path.join(path, "ADMISSIONS.csv"))
    df_diagnoses = pd.read_csv(os.path.join(path, "DIAGNOSES_ICD.csv"))

    df_diagnoses['ICD9_CODE'] = df_diagnoses['ICD9_CODE'].apply(extract_fun
    df_diagnoses['ICD9_CODE'] = df_diagnoses['ICD9_CODE'].map(code_dict)
    df_joined = df_admissions.merge(df_diagnoses, on=['HADM_ID', 'SUBJECT_I
    df_patient_admittee_ICD9 = df_joined[['SUBJECT_ID', 'ADMITTIME','ICD9_C

    patient_ids = []
    labels = []
    seq_data = []

    for name1, d in df_patient_admittee_ICD9.groupby('SUBJECT_ID'):
        patient_ids.append(name1)
        labels.extend(list(df_mortality["MORTALITY"][(df_mortality["SUBJECT
        subl = []
        d['ADMITTIME'] = pd.to_datetime(d['ADMITTIME'])
        for name2, subd in d.groupby('ADMITTIME'):
            subl.append(list(subd['ICD9_CODE']))
        seq_data.append(subl)

    return patient_ids, labels, seq_data
```

In [48]:
```
print("Building feature id map")
df_icd9 = pd.read_csv(os.path.join(PATH_DATA, "DIAGNOSES_ICD.csv"), usecols
code_dict = build_dict(df_icd9, extract_code)

print("Constructing train and test set")
ids, labels, seqs = build_dataset(PATH_DATA, code_dict, extract_code)
train_ids, test_ids, train_seqs, test_seqs, train_labels, test_labels = tra

print("Constructing validation set")
valid_ids, valid_labels, valid_seqs = build_dataset(PATH_VALIDATION, code_d
print("Completed!")
```

```
Building feature id map
Constructing train and test set
Constructing validation set
Completed!
```

In [49]:
```
ids[0:5] # patient IDs
```

Out[49]:
```
[17, 23, 34, 36, 61]
```

In [50]:
```
labels[0:5] # mortality
```

Out[50]:
```
[1, 1, 1, 1, 0]
```

In [51]:
```
seqs[0:5] # unique feature id
```

Out[51]:
```
[[[545, 336, 854, 171], [303, 375, 582, 336, 210, 525, 522, 171]],
 [[296, 293, 304, 872, 171, 287, 451, 279],
  [132, 242, 577, 304, 287, 171, 171, 872, 872, 857]],
 [[292, 308, 305, 307, 746, 306, 296, 782],
  [307, 308, 306, 296, 881, 308, 148, 258]],
 [[296, 293, 367, 287, 204, 393, 451, 852, 447],
  [747, 747, 297, 331, 745, 367, 296, 872, 287, 451, 393, 852],
  [408, 297, 382, 358, 746, 382, 296, 367, 287, 451, 199, 852]],
 [[113, 187, 186, 429, 184, 164, 748, 430, 498],
  [113,
   183,
   745,
   420,
   418,
   175,
   16,
   744,
   175,
   595,
   43,
   175,
   336,
   307,
   157,
   245,
   34]]]
```

2.2 Custom Pytorch Dataset

For each patient, I decided to use a matrix, rows represent different visits, jth column show the integer feature ID j. If matrix[i][j] == 1, it means that on ith visit, we get feature ID j.

In [52]:
```
# for each row, get the number of feature id.
def get_num_features(seqs):

    def get_flatten_list(ori_list, flatten_list = None):
        if flatten_list is None: flatten_list = []
        for i in ori_list:
            if isinstance(i, list): get_flatten_list(i, flatten_list)
            else: flatten_list.append(i)
        return flatten_list

    def get_max(list1):
        tmp = -1
        for i in list1:
            if i > tmp: tmp = i
        return tmp

    l = get_flatten_list(seqs)
    return get_max(l) + 1
```

In [53]:
```
# inherit from Dataset, represent matrixs.
class MyDataset(Dataset):
    def __init__(self, seqs, labels, num_features):

        self.labels = labels
        self.seqs = []
        def isNaN(num):
            return num != num
        for seq in seqs:
            new_seq = []
            for visit in seq:
                new_visit = [0] * int(num_features)
                for i in visit:
                    if not isNaN(i): new_visit[int(i)] = 1
                new_visit = np.asarray(new_visit)
                new_seq.append(new_visit)
            new_seq = np.asarray(new_seq)
            self.seqs.append(new_seq)

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, index):
        # returns will be wrapped as List of Tensor(s) by DataLoader
        return self.seqs[index], self.labels[index]
```

2.3 Generate mini-batches represented by 3D tensors

Generate mini-batches by defining `collate_fn` which is an argument of DataLoader constructor.

- mini-batches: batch_size * max_length * num_features. A min-batch consists of different patient's matrice
- If matrice in same batch have different number of rows, I will padding them with zero rows.

In [54]:
```
def collate_fn(batch):

    def getKey(item):
        return item[1]

    new_tuples = []
    seqs = []
    labels = []
    lengths = []
    max_length = -1
    num_features = len(batch[0][0][0])

    for b in batch:
        b_seqs = b[0]
        tmp_list = [b_seqs, len(b_seqs), b[1]]
        new_tuples.append(tuple(tmp_list))
        if len(b_seqs) > max_length:
            max_length = len(b_seqs)
    batch = sorted(new_tuples, key = getKey, reverse=True)

    for b in batch:
        b_seqs = b[0]
        labels.append(b[2])
        lengths.append(b[1])
        b_seqs = list(b_seqs)
        while len(b_seqs) < max_length:
            b_seqs.append([0] * num_features)
        seqs.append(torch.Tensor(b_seqs))
    seqs_tensor = torch.stack(seqs, 0)
    lengths_tensor = torch.LongTensor(lengths)
    labels_tensor = torch.LongTensor(labels)

    return (seqs_tensor, lengths_tensor), labels_tensor
```

3. Build RNN model

- Only need to define the forward function, the backward function will be generate automatically

In [55]:
```
class MyNet(nn.Module):
    def __init__(self, dim_input):
        super(MyNet, self).__init__()
        self.fc1 = nn.Linear(dim_input, 32)
        self.rnn = nn.GRU(input_size=32, hidden_size=16, num_layers=2, batc
        self.fc2 = nn.Linear(16, 2)
        self.tanh = nn.Tanh()

    def forward(self, input_tuple):
        seqs, lengths = input_tuple
        seqs = self.fc1(seqs)
        seqs = F.tanh(seqs)
        seqs = pack_padded_sequence(seqs, lengths, batch_first=True)
        seqs, _ = self.rnn(seqs)
        seqs, _ = pad_packed_sequence(seqs, batch_first=True)
        new_seqs = []
        for i in range(0, len(seqs)):
            new_seqs.append(seqs[i][lengths[i] - 1])
        new_seqs = torch.stack(new_seqs, 0)
        seqs = self.fc2(new_seqs)
        return seqs
```

4. Training and validation process

In [56]:
```
import os
import time

class MyAverageGenerator():
    def __init__(self):
        self.val = 0
        self.average = 0
        self.sum = 0
        self.count = 0

    def update(self, val, size = 1):
        self.val = val
        self.sum += val * size
        self.count += size
        self.average = self.sum / self.count

'''
Reference: https://github.com/cse6250/SepsisPrediction/blob/1b732548e263a29
'''

def get_batch_accuracy(output, target):

    with torch.no_grad():

        batch_size = target.size(0)
        _, pred = output.max(1)
        correct = pred.eq(target).sum()

        return correct * 100.0 / batch_size

def train(model, device, data_loader, criterion, optimizer, epoch, print_fr
    batch_time = MyAverageGenerator()
    data_time = MyAverageGenerator()
    losses = MyAverageGenerator()
    acc = MyAverageGenerator()

    model.train()

    end = time.time()
    for i, (input, target) in enumerate(data_loader):
        # measure data loading time
        data_time.update(time.time() - end)

        if isinstance(input, tuple):
            input = tuple([e.to(device) if type(e) == torch.Tensor else e f
        else:
            input = input.to(device)
        target = target.to(device)

        optimizer.zero_grad()
        output = model(input)
        loss = criterion(output, target)
        assert not np.isnan(loss.item()), 'Model diverged with loss = NaN'

        loss.backward()
        optimizer.step()
```

```
        # measure elapsed time
        batch_time.update(time.time() - end)
        end = time.time()

        losses.update(loss.item(), target.size(0))
        acc.update(get_batch_accuracy(output, target).item(), target.size(0

        if i % print_freq == 0:
            print('Epoch: [{0}][{1}/{2}]\t'
                  'Time ({batch_time.average:.3f})\t'
                  'Data ({data_time.average:.3f})\t'
                  'Loss ({loss.average:.4f})\t'
                  'Accuracy ({acc.average:.3f})'.format(epoch, i, len(data_

    return losses.average, acc.average


def evaluate(model, device, data_loader, criterion, print_freq=10):
    batch_time = MyAverageGenerator()
    losses = MyAverageGenerator()
    acc = MyAverageGenerator()

    results = []

    model.eval()

    with torch.no_grad():
        end = time.time()
        for i, (input, target) in enumerate(data_loader):

            if isinstance(input, tuple):
                input = tuple([e.to(device) if type(e) == torch.Tensor else
            else:
                input = input.to(device)
            target = target.to(device)

            output = model(input)
            loss = criterion(output, target)

            # measure elapsed time
            batch_time.update(time.time() - end)
            end = time.time()

            losses.update(loss.item(), target.size(0))
            acc.update(get_batch_accuracy(output, target).item(), target.si

            y_true = target.detach().to('cpu').numpy().tolist()
            y_pred = output.detach().to('cpu').max(1)[1].numpy().tolist()
            results.extend(list(zip(y_true, y_pred)))

            if i % print_freq == 0:
                print('Test: [{0}/{1}]\t'
                      'Time ({batch_time.average:.3f})\t'
                      'Loss ({loss.average:.4f})\t'
                      'Accuracy ({acc.average:.3f})'.format(i, len(data_loa

    return losses.average, acc.average, results
```

In [57]:
```
torch.manual_seed(0)
NUM_EPOCHS = 20
BATCH_SIZE = 32
USE_CUDA = False
NUM_WORKERS = 0
```

In [58]:
```
num_features = get_num_features(train_seqs)

train_dataset = MyDataset(train_seqs, train_labels, num_features)
valid_dataset = MyDataset(valid_seqs, valid_labels, num_features)
test_dataset = MyDataset(test_seqs, test_labels, num_features)

train_loader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shu
valid_loader = DataLoader(dataset=valid_dataset, batch_size=BATCH_SIZE, shu
# batch_size for the test set should be 1 to avoid sorting each mini-batch
test_loader = DataLoader(dataset=test_dataset, batch_size=1, shuffle=False,
```

In [59]:
```
model = MyNet(num_features)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)

device = torch.device("cuda" if torch.cuda.is_available() and USE_CUDA else
model.to(device)
criterion.to(device)
```

Out[59]:
```
CrossEntropyLoss()
```

In [60]:
```python
best_val_acc = 0.0
train_losses, train_accuracies = [], []
valid_losses, valid_accuracies = [], []
for epoch in range(NUM_EPOCHS):
    train_loss, train_accuracy = train(model, device, train_loader, criteri
    valid_loss, valid_accuracy, valid_results = evaluate(model, device, val

    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    train_accuracies.append(train_accuracy)
    valid_accuracies.append(valid_accuracy)

    is_best = valid_accuracy > best_val_acc  # keep the best model
    if is_best:
        best_val_acc = valid_accuracy
        torch.save(model, os.path.join(PATH_OUTPUT, "MyNet.pth"))
```

```
ccuracy (78.965)
Epoch: [16][50/132]     Time (0.021)    Data (0.013)    Loss (0.4544)   A
ccuracy (79.228)
Epoch: [16][60/132]     Time (0.021)    Data (0.013)    Loss (0.4479)   A
ccuracy (80.020)
Epoch: [16][70/132]     Time (0.021)    Data (0.013)    Loss (0.4473)   A
ccuracy (79.621)
Epoch: [16][80/132]     Time (0.022)    Data (0.013)    Loss (0.4542)   A
ccuracy (79.167)
Epoch: [16][90/132]     Time (0.022)    Data (0.013)    Loss (0.4550)   A
ccuracy (79.087)
Epoch: [16][100/132]    Time (0.022)    Data (0.013)    Loss (0.4521)   A
ccuracy (79.022)
Epoch: [16][110/132]    Time (0.023)    Data (0.014)    Loss (0.4520)   A

ccuracy (78.970)
Epoch: [16][120/132]    Time (0.023)    Data (0.013)    Loss (0.4516)   A
ccuracy (79.055)
Epoch: [16][130/132]    Time (0.024)    Data (0.014)    Loss (0.4531)   A
ccuracy (79.008)
Test: [0/24]    Time (0.021)    Loss (0.5329)   Accuracy (65.625)
```

5. Plot best model

In [22]:
```
import matplotlib
matplotlib.use("TkAgg")
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

def plot_learning_curves(train_losses, valid_losses, train_accuracies, vali
    print("******************************Plot learning curves**************
    x = len(train_losses)
    x = np.arange(x)
    plt.figure()
    plt.plot(x, train_losses, label = "train_losses")
    plt.plot(x, valid_losses, label = "valid_losses" )
    plt.title("Loss Curve")
    plt.xlabel("epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.savefig(title + "_loss_curve.png")
    plt.show()

    print("******************************Plot accuracy curves**************
    x = len(valid_accuracies)
    x = np.arange(x)
    plt.figure()
    plt.plot(x, train_accuracies, label = "train_accuracies")
    plt.plot(x, valid_accuracies, label = "valid_accuracies")
    plt.title("Accuracy Curve")
    plt.xlabel("epoch")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.savefig(title + "_acc_curve.png")
    plt.show()


def plot_confusion_matrix(results, class_names, title):
    print("******************************Plot confusion maxtrix************
    y_true, y_pred = zip(*results)
    cm = confusion_matrix(y_true, y_pred)
    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    print(cm)
    fig, ax = plt.subplots()
    im = ax.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
    ax.figure.colorbar(im, ax=ax)

    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           xticklabels=class_names, yticklabels=class_names,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")
    fmt = '.2f'
    thresh = cm.max() / 2
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
```

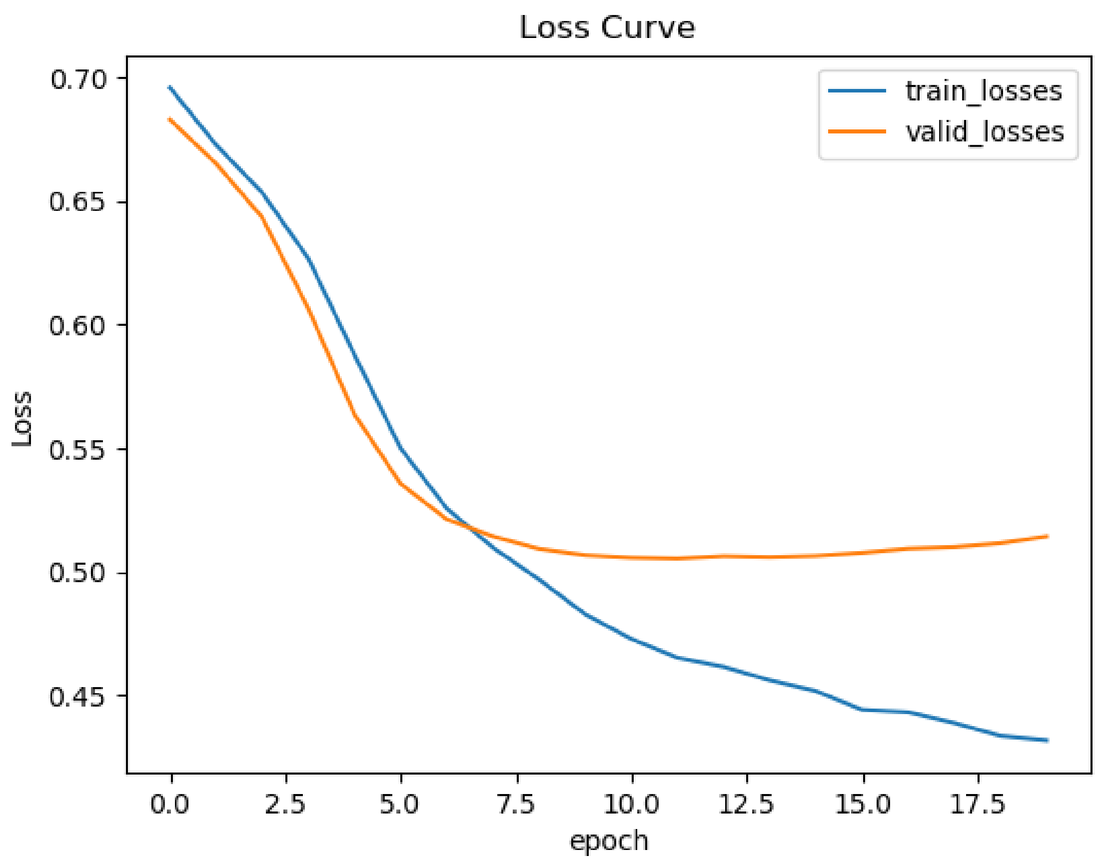
```
                        ha='center', va='center', color='white' if cm[i, j] > t
    fig.tight_layout()
    plt.savefig(title + "_cm.png")
    plt.show()
```

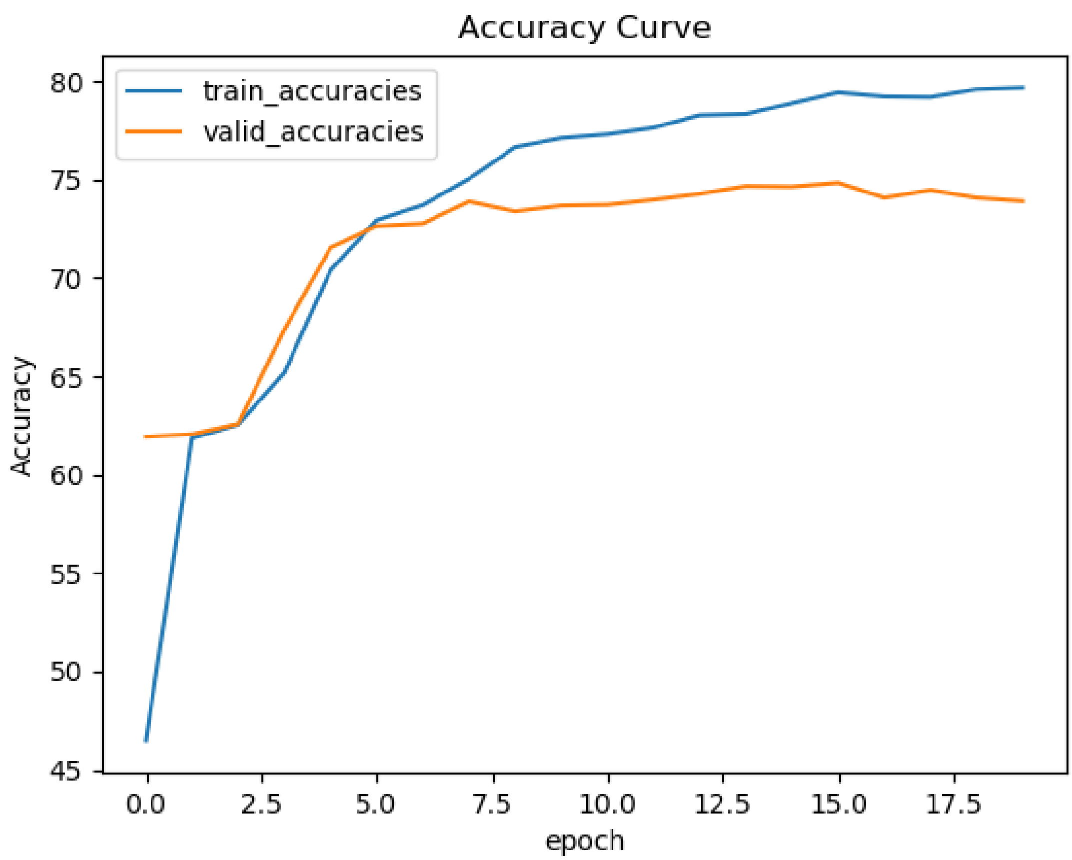
In [31]:
```
best_model = torch.load(os.path.join(PATH_OUTPUT, "MyNet.pth"))
plot_learning_curves(train_losses, valid_losses, train_accuracies, valid_ac
```

```
****************************Plot learning curves********************

An exception has occurred, use %tb to see the full traceback.

SystemExit: 0
```

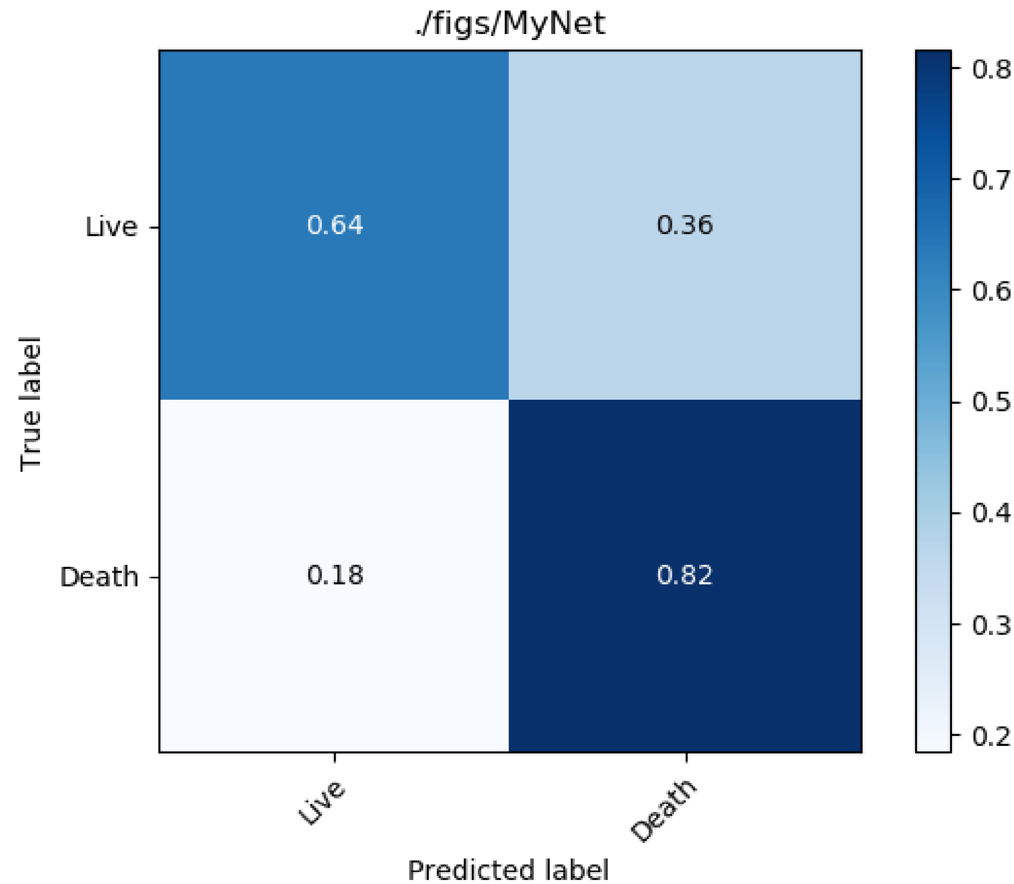




In [ ]:
```
class_names = ['Live','Death']
plot_confusion_matrix(valid_results, class_names, os.path.join(PATH_OUTPUT,
```

```
****************************Plot confusion maxtrix*********************
[[0.6360424  0.3639576 ]
 [0.18471338 0.81528662]]
```

- Precision: 0.64
- Recall: 0.78


./figs/MyNet
Live
Death
0.64
0.36
0.18
0.82
True label
Predicted label
Live
Death
0.8
0.7
0.6
0.5
0.4
0.3
0.2

6. Predict

In [27]:
```
def predict_mortality(model, device, data_loader):
    model.eval()
    model.to(device)
    probas = []
    trues = []
    with torch.no_grad():
        for i, (input, target) in enumerate(data_loader):

            if isinstance(input, tuple):
                input = tuple([e.to(device) if type(e) == torch.Tensor else
            else:
                input = input.to(device)

            output = model(input)
            output = nn.Sigmoid()(output)

            if output[0][0] < output[0][1]:
                probas.append(1)
            else:
                probas.append(0)
            trues.append(target)
    acc = accuracy_score(trues, probas)
    print('The accuracy of test dataset is {}'.format(acc))
    return probas


test_prob = predict_mortality(best_model, device, test_loader)
```

```
The accuracy of test dataset is 0.7469194312796209
```

In [28]:
```
def make_report(list_id, list_prob, path):
    if len(list_id) != len(list_prob):
        raise AttributeError("ID list and Probability list have different l

    os.makedirs(path, exist_ok=True)
    output_file = open(os.path.join(path, 'my_predictions.csv'), 'w')
    output_file.write("SUBJECT_ID,MORTALITY\n")
    for pid, prob in zip(list_id, list_prob):
        output_file.write("{},{}\n".format(pid, prob))
    output_file.close()
```

In [29]:
```
make_report(test_ids, test_prob, PATH_OUTPUT)
num_test_patient = len(test_ids)
```

In [30]:
```
num_test_patient
```

Out[30]:
```
1055
```