Multiple instance learning to predict survival from multiplex immunofluorescence images

Kim Zhirou Li

1. Introduction

A wide range of immunological biomarkers in tissue samples were collected at the Center for Immuno-Oncology at Dana-Farber. Among all these cancer biomarkers, Non-Small Cell Lung Cancer (NSCLC) biomarker was used in this project because NSCLC has the largest amount of data, which is a key prerequisite for training valid machine learning models. The big goal of this project intends to explore the association between NSCLC biomarkers and patient outcomes. I approach this goal by two sub-goals. The first sub-goal is developing some different machine learning models to see whether machine learning algorithms can help to predict patients' 2 years survival (0 or 1). The second sub-goal is checking whether adding spatial information, which describes where the embedding is in the original medical images, to the input of machine learning models could help improve models' performance, which consequently contributes to predict patients' survival.

NSCLC data were firstly processed by computer-assisted digital quantification and manual Region of Interest (ROI) selection. Then, biomarkers from NSCLC data were identified in an interpretable manner. In specific, the unit used is cells and their spatial arrangement. Because of the nature of cells is their arrangement in a non-Euclidean space, Spatial Cellular Graphical Modelling were used processed the cells into embeddings.

Those ROI embeddings are the data that I received as input to different machine learning models. Here is the comprehensive description of them. Among the data, there are 507 cases in total and 1841 Inner tumor ROIs were generated from the cases. 1376 pieces of data are split for development (training and validation) and 465 pieces are for testing. Each ROI embedding is a 2- dimensional data with different first dimension and same second dimension. Thus, I started to explore the dataset with classic Multiple Instance Learning (MIL) and then tried simple-attention modules and a small transformer with multi-head attention. The graphs showing the distribution of number of instances in a bag/ROI and the number of 0/1 labels of each ROI are in Appendix.

2. Methods

2.0 Process different length in each ROI and spatial information

Initially, ROI embeddings and corresponding labels are at twp different files, which can be matched through aq_id. So I created two functions to extract aq_id, embeddings, labels and spatial information from the two files and stored them in a dictionary. This project faces two major difficulties, which are the different length of each bag (ROI) and embed spatial information to ROI embeddings.

In terms of tackling the different length of each bag (ROI), I explored three different ways. a. Use classic MIL methods that aggregate different number of instances in each bag into 1 instance so that each ROI has the same dimension of (1,32). (Refer to 2.1)* b. Pad all ROIs' lengths to the maximum length with 0 (Refer to 2.2 and 2.3). c. Use Attention module, which can tolerate different bags' lengths in different batches because it will output a 132 matrix (Refer to 2.2 and 2.3).

In terms of processing spatial information, I explored two different ways. d. Concatenate the spatial information to ROI embedding so that the second dimension (feature) of embedding is extended from 32 to 34 (Refer to 2.2 and 2.3). e. Use neural network to project the 2 dimensional spatial information to 32 and add spatial embedding to ROI embedding so that the second dimension (feature) of the overall embedding remain 32 (Refer to 2.2 and 2.3).

2.1 Classic MIL aggregation and ML

The classic way to approach multiple instance learning is by aggregating the all the instances in a bag into 1 instance using different pooling methods. Here I explored 4 different data indexes, i.e. 'mean': [np.mean], 'var': [np.var], 'kurtosis': [kurtosis] and 'max':[np.max]. I combine them in 12 ways exhaustively and use Logistic Regression with 3-fold cross validation to calculate their AUROC and Accuracy. The reason I chose Logistic Regression is because this is a binary classification problem and logistic regression can help to project values to range between 0 and 1.

2.2 Simple attention module

This part employed a weighted average of instances, determined by a neural network, with weights that sum to 1, ensuring invariance to bag size. Let (h_k) be a bag of k embeddings, and below is the function for MIL pooling:

$$z = \sum_{k=1}^{K} a_k h_k$$

where

$$a_k = \frac{\exp\{w^T \tanh(Vh_k)\}}{\sum_{j=1}^{K} \exp\{w^T \tanh(Vh_j)\}}$$

Here (a_k) is the attention weights after "softmax" for each (h_k).

A bag of embeddings with dimension (b, n, l) processed after this simple attention module will be (b,1,l) as the attention module will learn the attention weights for each of the n instance and apply the attention weights to each of the n instance.

2.3 Transformer with multi-head attention

I also employed a classic transformer architecture with multi-heads attention. Below are the ways to calculate attention matrices, which is different than the simple attention module above. This time the attention weights are learnt from Q and K, which are both linear projection with different parameters from input, divided by the length of Q, L, V and input (which are all same). Same as before, this attention mechanism can be applied in Multiple instance learning because no matter what n will be in each batch, the output of the bag will always have a dimension of (b,1,l).

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Transformer also uses a multi-head mechanism where h heads or h attentions will be generated in parallel. In each head, the length of input, i.e. d_k will be divided by h. In this case, when concatenate all heads at the end of each attention block, the length of input can be converted back to original d_k.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O$$

where

$$\text{head}_i = \text{Attention}(QW_{Q_i}, KW_{K_i}, VW_{V_i})$$

3. Results

All of the results are evaluated with a 3-fold cross validation and AUROC is the main metric we are interested in because our data set is small and data is not very consistent.

3.1 Classic MIL aggregation and ML Results

3.1.1 Logistic regression without spatial information

Logistic regression without spatial information achieved a max AUROC of 0.660 as shown in Appendix.

3.1.2 Result with spatial information processed by d method

Logistic regression with spatial information embedded using d method achieved a lower AUROC in general with a max of 0.657. Overall speaking, mean_kurtosis performs well in both cases with a consistent AUROC of 0.654.

3.2 Simple attention module Results

3.2.1 Simple attention module without spatial information

For the last epoch in each of the 3 fold, the average AUROC is 0.7197, higher than any method of the logistic regression in 3.1. The last epoch in each of the 3 fold achieves a fairly stable AUROC, which ranges from 0.7124 and 0.7303. For the AUROC trend in 45 epochs in each of the 3 cross-validations, we can see the AUROC in first fold increased dramatically from start while in the second and third fold, the AUROC was very high at start and maintained high since.

3.2.2 Simple attention module with spatial information by d

For the last epoch in each of the 3 fold, the average AUROC is 0.6791, showing that concatenating spatial information does not help improve model performance with simple attention module. The plot of the last epoch in each of the 3 fold ranges from 0.6547 to 0.6791. For the AUROC trend in 45 epochs in each of the 3 cross-validations, it is shown that all of the folds illustrated an increasing trend in AUROC as epochs go.

3.2.3 Simple attention module with spatial information by e

For the last epoch in each of the 3 fold, the average AUROC is 0.7197, the same performance as not adding any spatial data. For the AUROC trend in 45 epochs in each of the 3 cross-validations, AUROC in first fold increased dramatically from start while in the second and third fold, the AUROC was very high at start and maintained high since.

3.3 Transformer with multi-head attention Results

3.3.1 Transformer with multi-head attention without spatial information

For the last epoch in each of the 3 fold, the average AUROC is 0.7142, not comparable to single-attention module. In addition, transformer needs to be trained on GPU with high computational comsumptions and time. Single-attention module seems to be a more reasonable choice in this project. The plot of the last epoch in each of the 3 fold ranges from 0.70 to 0.734. From the AUROC trend in 45 epochs in each of the 3 cross-validations, we can tell all of the folds achieves a stable AUROC across epochs but not the highest AUROC. ### 3.3.2 Transformer with multi-head attention with spatial information by d For the last epoch in each of the 3 fold, the average AUROC is 0.6762. From the plot of the last epoch in each of the 3 fold, the values range from 0.63 to 0.72. From the AUROC trend in 45 epochs in each of the 3 cross-validations, the trends is very fluctuating and do not show any trend or pattern, meaning the training is not valid.

3.3.3 Transformer with multi-head attention with spatial information by e

For the last epoch in each of the 3 fold, the average AUROC is 0.6888. For the AUROC trend in 45 epochs in each of the 3 cross-validations, all folds gradually achieves a balanced and stable AUROC, which are not highest AUROC.

4. Conclusion

This project answered two questions proposed at the beginning. First, I found out that machine learning models can be helpful to predict a patient's 2-year survival with valid performance. Among all the models experimented, simple-attention module performed the best with an average AUROC of 0.72. It is also the model that needs low computational resources so that it can be ran on personal laptop instead of a GPU. Second, I also tried two ways to embed spatial information and the results showed that these two ways of adding spatial information cannot help improve the model's performance. More spatial information adding methods need to be explored.

5. Appendix

Extra Findings:

a. Due to small dataset and the inconsistencies between training and validation data, I find that using 3-fold cross validation will be rational in the case more data would be used to moderately prevent the training and validation data heterogeneity.

b. In Simple-attention models, setting a relatively large epoch is necessary since data tend to be fallen into a "lowland" and keeping decreasing training loss by increasing epoch number could help at least to reach a local minimum.

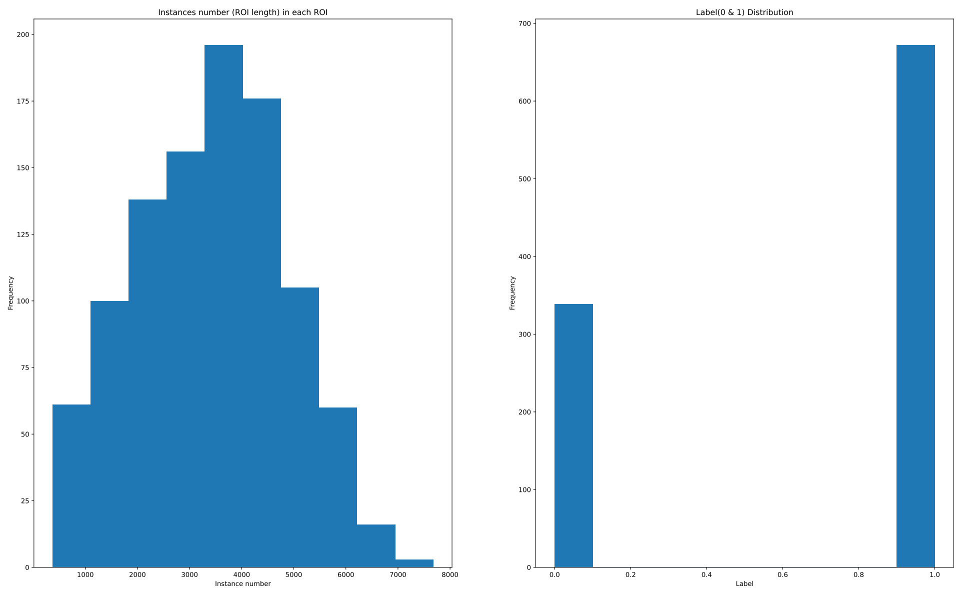
Code for 1

```
import math
import numpy as np
import os
import pandas as pd
import pickle
import random
import json
import matplotlib.pyplot as plt
import sys
import torch
import torch.nn as nn
import torch.nn.functional as F

from torch.nn.utils import clip_grad_norm_
# Baseline aggregation models with combinations of 4 methods
from sklearn.metrics import roc_auc_score
from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler, TensorDataset
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import balanced_accuracy_score
from sklearn.linear_model import LogisticRegression
import numpy as np
import pandas as pd
from scipy.stats import kurtosis
from functools import partial
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, balanced_accuracy_score, roc_curve, precision_
from scipy.stats import mode
```

```python
# !! Change the data_root Import data
data_root = '/Users/kimli/Desktop/260fp/data_new/'

# each roi's spatial position and GNN embeddings
with open(os.path.join(data_root, 'development_modified_embeddings.pkl'), 'rb') as file:
    emb_dict = pickle.load(file)


# !! Change the data_root Import each roi's label
label_df=pd.read_csv("/Users/kimli/Desktop/260fp/data_new/nsclc_roi_labels.csv")



def grab_label(label_df, aq_id, label_name):

    temp_label = label_df.loc[label_df['aq_id']==aq_id][label_name].values[0]
    if math.isnan(temp_label):
        return None
    else:
        return temp_label


def generate_instance_dict(data_dict, label_df, aq_ids=None, split_half=False):

    # instance dict: each entry are the subgraph features and label for one ROI
    # only contains ROIs for which a valid label exists
    instance_dict = {'labels': list(),
                    'aq_ids': list(),
                     'embedbags':list(),
                     'spatialembedbags':list()}

    if aq_ids is None:
        aq_ids = list(data_dict.keys())

    for k in aq_ids:
        # grab label first
        temp_label = grab_label(label_df, k, '2y_survival')

        if temp_label is not None:
            if split_half:
                # split each ROI into two bags (random sampling of subgraphs/instances)
                temp_bagspatial = np.asarray(data_dict[k][0])

                temp_bagembed = np.asarray(data_dict[k][1])

                n_instances = temp_bagembed.shape[0] # number of subgraphs in the ROI
                idx = np.arange(n_instances)
                # random.shuffle(idx)

                # split the bag and add to dictionary
                subbagembed_1 = temp_bagembed[idx[:int(n_instances/2)],:]
                subbagspatial_1=temp_bagspatial[idx[:int(n_instances/2)],:]
                instance_dict['embedbags'].append(torch.tensor(np.array(subbagembed_1)).fl
                instance_dict['spatialembedbags'].append(torch.cat([torch.tensor(np.asarra
                instance_dict['labels'].append(temp_label)
                instance_dict['aq_ids'].append(k)
                subbagembed_2 = temp_bagembed[idx[int(n_instances/2):],:]
                subbagspatial_2=temp_bagspatial[idx[int(n_instances/2):],:]
                instance_dict['embedbags'].append(torch.tensor(np.array(subbagembed_2)).fl
                instance_dict['spatialembedbags'].append(torch.cat([torch.tensor(np.asarra
                instance_dict['labels'].append(temp_label)
                instance_dict['aq_ids'].append(k)

            else:
                instance_dict['labels'].append(temp_label)
                instance_dict['aq_ids'].append(k)
                instance_dict['embedbags'].append(torch.tensor(np.array(data_dict[k][1])).
                instance_dict['spatialembedbags'].append(torch.cat([torch.tensor(np.asarra

    return instance_dict


def getaqids(indices):
  temp=[]
  for i in range(len(indices)):
    temp.append(list(data_dict.keys())[indices[i]])
  return temp


data_dict = emb_dict['embedding_dict']

# Create a list of indices from 0 to 1357 (assuming 0-based indexing) 1358 ROIs
```

```
total_samples = len(data_dict.keys())
indices = list(range(total_samples))

all_indices=indices
all_aqids=getaqids(all_indices)
all_instance_dict=generate_instance_dict(data_dict, label_df, aq_ids=all_aqids, split_half
```

```
# Visualize distribution of bag length(instances number in each bag) in train, vali
fig, axes = plt.subplots(1, 2, figsize=(25,15))
axes[0].hist([len(bag) for bag in all_instance_dict['embedbags']])
axes[0].set_xlabel('Instance number')
axes[0].set_ylabel('Frequency')
axes[0].set_title('Instances number (ROI length) in each ROI')
axes[1].hist(all_instance_dict['labels'])
axes[1].set_xlabel('Label')
axes[1].set_ylabel('Frequency')
axes[1].set_title('Label(0 & 1) Distribution')


plt.show()
```



Code & Result for 3.1.1

```
def extract_summary_stats(all_instance_dict, summary_funcs=[np.mean, np.var, kurtosis]):
    summary_stats = []

    for summary_func in summary_funcs:
        # Calculate the summary statistic for each bag
        bag_stats = [summary_func(np.array(bag), axis=0) for bag in all_instance_dict["emb
        # Stack the bag statistics into a 2D array
        summary_stats.append(np.vstack(bag_stats))

    # Concatenate the summary statistics horizontally (axis=1)
    concatenated_stats = np.hstack(summary_stats)

    return concatenated_stats



summary_statistics_list = {'mean': [np.mean], 'var': [np.var], 'kurtosis': [kurtosis], 'ma
                           'mean_var': [np.mean, np.var], 'mean_kurtosis': [np.mean, kurto
                           'mean_var_kurtosis': [np.mean, np.var, kurtosis],'mean_var_max'
                           'mean_var_kurtosis_max': [np.mean, np.var, kurtosis,np.max]}




def get_roc_auc(y_true, y_pred):
    fpr, tpr, _  = roc_curve(y_true, y_pred)
    roc_auc = auc(fpr, tpr)
    return roc_auc


def get_pr_auc(y_true, y_pred):
    precision, recall, _ = precision_recall_curve(y_true, y_pred, pos_label=1)
    pr_auc = auc(recall, precision)
    return pr_auc


def get_accuracy(y_true, y_pred):
    return balanced_accuracy_score(y_true, y_pred)

df = []
```

```
skf = StratifiedKFold(n_splits=3, shuffle=True, random_state=25)

for summary_statistics, stats_func in summary_statistics_list.items():
    roc_auc_list = []
    pr_auc_list = []
    balanced_accuracy_list = []

    for train_index, val_index in skf.split(all_instance_dict["embedbags"], all_instance_d
        X_train_stats = extract_summary_stats(all_instance_dict, stats_func)
        model = LogisticRegression(max_iter=10000).fit(X_train_stats[train_index], np.asar
        y_val_pred = model.predict(X_train_stats[val_index])

        roc_auc = get_roc_auc(np.asarray(all_instance_dict['labels'])[val_index], y_val_pr
        pr_auc = get_pr_auc(np.asarray(all_instance_dict['labels'])[val_index], y_val_pred
        balanced_accuracy = balanced_accuracy_score(np.asarray(all_instance_dict['labels']

        roc_auc_list.append(roc_auc)
        pr_auc_list.append(pr_auc)
        balanced_accuracy_list.append(balanced_accuracy)

    avg_roc_auc = np.mean(roc_auc_list)
    avg_pr_auc = np.mean(pr_auc_list)
    avg_balanced_accuracy = np.mean(balanced_accuracy_list)

    df.append((summary_statistics, avg_roc_auc, avg_pr_auc, avg_balanced_accuracy))

df = pd.DataFrame(df, columns=['Model', 'ROC AUC', 'PR AUC', 'Balanced Accuracy']).sort_va
print(df)
```

```
                   Model   ROC AUC    PR AUC  Balanced Accuracy
13  mean_var_kurtosis_max  0.660082  0.854747           0.660082
5           mean_kurtosis  0.657145  0.854088           0.657145
10      mean_var_kurtosis  0.651937  0.851210           0.651937
7                mean_max  0.650607  0.854233           0.650607
11           mean_var_max  0.649132  0.853795           0.649132
0                    mean  0.647005  0.855248           0.647005
12       max_var_kurtosis  0.644602  0.849635           0.644602
4                mean_var  0.640348  0.852680           0.640348
8            max_kurtosis  0.639433  0.847737           0.639433
9                 var_max  0.624045  0.845368           0.624045
3                     max  0.623288  0.844567           0.623288
6            var_kurtosis  0.587119  0.832507           0.587119
```

```
2              kurtosis  0.579784  0.831550           0.579784
1                   var  0.524264  0.835183           0.524264
```

Code & Result for 3.1.2

```
def extract_summary_stats(all_instance_dict, summary_funcs=[np.mean, np.var, kurtosis]):
    summary_stats = []

    for summary_func in summary_funcs:
        # Calculate the summary statistic for each bag
        bag_stats = [summary_func(np.array(bag), axis=0) for bag in all_instance_dict["spa
        # Stack the bag statistics into a 2D array
        summary_stats.append(np.vstack(bag_stats))

    # Concatenate the summary statistics horizontally (axis=1)
    concatenated_stats = np.hstack(summary_stats)

    return concatenated_stats



summary_statistics_list = {'mean': [np.mean], 'var': [np.var], 'kurtosis': [kurtosis], 'ma
                           'mean_var': [np.mean, np.var], 'mean_kurtosis': [np.mean, kurto
                           'mean_var_kurtosis': [np.mean, np.var, kurtosis],'mean_var_max'
                           'mean_var_kurtosis_max': [np.mean, np.var, kurtosis,np.max]}



def get_roc_auc(y_true, y_pred):
    fpr, tpr, _  = roc_curve(y_true, y_pred)
    roc_auc = auc(fpr, tpr)
    return roc_auc


def get_pr_auc(y_true, y_pred):
    precision, recall, _ = precision_recall_curve(y_true, y_pred, pos_label=1)
    pr_auc = auc(recall, precision)
    return pr_auc


def get_accuracy(y_true, y_pred):
```

```python
    return balanced_accuracy_score(y_true, y_pred)

df = []
skf = StratifiedKFold(n_splits=3, shuffle=True, random_state=25)

for summary_statistics, stats_func in summary_statistics_list.items():
    roc_auc_list = []
    pr_auc_list = []
    balanced_accuracy_list = []

    for train_index, val_index in skf.split(all_instance_dict["spatialembedbags"], all_ins
        X_train_stats = extract_summary_stats(all_instance_dict, stats_func)
        model = LogisticRegression(max_iter=10000).fit(X_train_stats[train_index], np.asar
        y_val_pred = model.predict(X_train_stats[val_index])

        roc_auc = get_roc_auc(np.asarray(all_instance_dict['labels'])[val_index], y_val_pr
        pr_auc = get_pr_auc(np.asarray(all_instance_dict['labels'])[val_index], y_val_pred
        balanced_accuracy = balanced_accuracy_score(np.asarray(all_instance_dict['labels']

        roc_auc_list.append(roc_auc)
        pr_auc_list.append(pr_auc)
        balanced_accuracy_list.append(balanced_accuracy)

    avg_roc_auc = np.mean(roc_auc_list)
    avg_pr_auc = np.mean(pr_auc_list)
    avg_balanced_accuracy = np.mean(balanced_accuracy_list)

    df.append((summary_statistics, avg_roc_auc, avg_pr_auc, avg_balanced_accuracy))

df = pd.DataFrame(df, columns=['Model', 'ROC AUC', 'PR AUC', 'Balanced Accuracy']).sort_va
print(df)
```

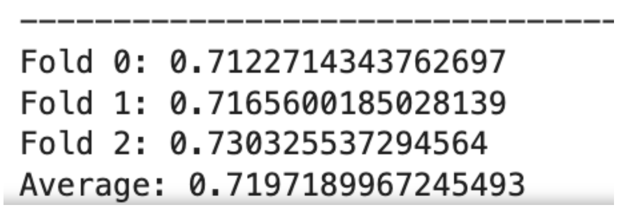
```
                 Model   ROC AUC    PR AUC  Balanced Accuracy
7             mean_max  0.660918  0.857396           0.660918
5        mean_kurtosis  0.651246  0.852076           0.651246
8         max_kurtosis  0.642383  0.848767           0.642383
0                 mean  0.636655  0.851307           0.636655
3                  max  0.629214  0.847153           0.629214
2             kurtosis  0.577539  0.829842           0.577539
4             mean_var  0.576927  0.834054           0.576927
13  mean_var_kurtosis_max  0.560011  0.830495           0.560011
1                  var  0.559438  0.835984           0.559438
```

```
6           var_kurtosis  0.559438  0.835984           0.559438
10      mean_var_kurtosis  0.558602  0.832263           0.558602
11           mean_var_max  0.558562  0.830979           0.558562
9                 var_max  0.557818  0.830527           0.557818
12       max_var_kurtosis  0.557818  0.830527           0.557818
```

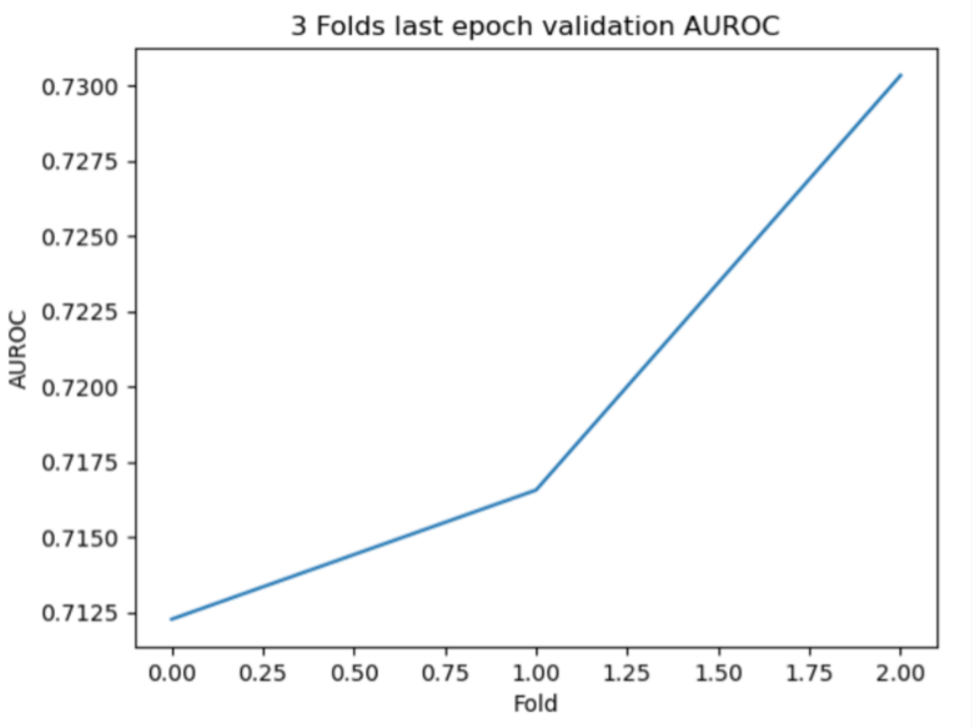
Code for 3.2

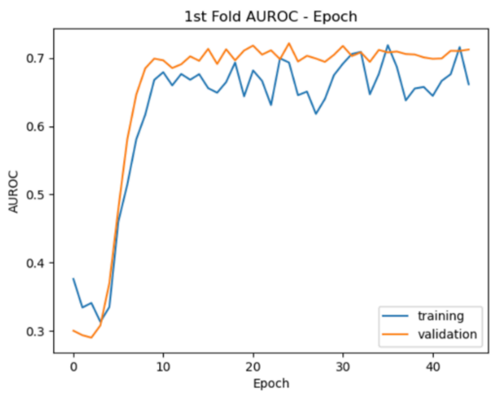
```
# Model: Simple attention module

# class Attention(nn.Module):
#     def __init__(self,L):
#
#         super(Attention, self).__init__()
#         self.embedspatial=nn.Sequential(
#         nn.Linear(2, 32),
#         nn.LayerNorm(32))
#
#         self.L = L # number of input features
#         self.D = int(self.L/4) # dimension for attention module
#         self.K = 1 # aggregated dimension (number of "instances" after attention) - outp
#
#         self.attention = nn.Sequential(
#             nn.Linear(self.L, self.D),
#             nn.Tanh(),
#             nn.Linear(self.D, self.K)
#         )
#
#         '''self.classifier = nn.Sequential(
#             nn.Linear(self.L*self.K, 1),
#             nn.Linear(self.L*self.K, 1)
#             nn.Sigmoid()
#         )'''
#
#         if self.L > 256:
#             self.classifier = nn.Sequential(
#                     nn.Linear(self.L*self.K, self.L),
#                     nn.LeakyReLU(),
#                     nn.Linear(self.L, self.L),
#                     nn.LeakyReLU(),
#                       nn.Linear(self.L, 1)
#               )
#
#           else:
#               '''self.classifier = nn.Sequential(
#                       nn.Linear(self.L*self.K, self.L),
#                       nn.LeakyReLU(),
#                       nn.Linear(self.L, 1))'''
#               self.classifier = nn.Linear(self.L, 1)
#
#       def forward(self, H):
#           # H: BxNxL
#           B = H.size()[0]
#           N = H.size()[1]
#
#           #!! With embedded Spatial information
#           # H= self.embedspatial(H[:,:,:2])+H[:,:,2:]
#
#           #!! With concat spatial information, n_embed =34
#           H = H
#
#           #!! Without spatial information
#           # H=H[:,:,2:]
#
#           # Pay attention to keep the output as BxNx1
#           A = self.attention(H).transpose(-1, -2) # Bx1xN (weights for each instance)
#           # generate an attention mechanism that doesn't do anything (compute the mean)
#           # A = torch.ones((B,1,N)).to(H.device) # .to(device)
#           A = torch.softmax(A, dim=2) # normalize along N (instance weights sum up to 1)
#
#           # multiply H with A (apply attention)
#           # transpose H from BxNxL to BxLxN for proper multiplication
#           M = A * H.transpose(1, 2) # BxLxN (weighted along N)
#
#           # sum over N dimension
#           M = torch.sum(M, dim=2, keepdim=True) # BxLx1
#
#           # Now, to match the classifier input shape, we reshape M from BxLx1 to Bx1xL
#           M = M.transpose(1, 2) # Bx1xL
#
#           Y_prob = self.classifier(M.view(M.size(0), -1))
#         # Y_hat = torch.ge(Y_prob, 0.5).float()
#
#
#         return Y_prob
```

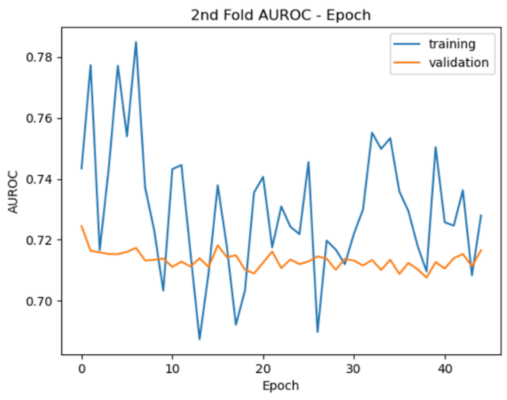
Results for 3.2.1

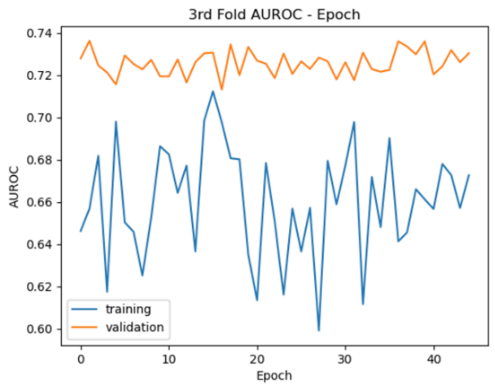


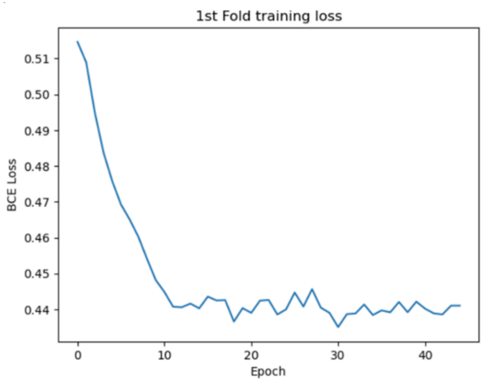
```
--------------------------------
Fold 0: 0.7122714343762697
Fold 1: 0.7165600185028139
Fold 2: 0.730325537294564
Average: 0.7197189967245493
```

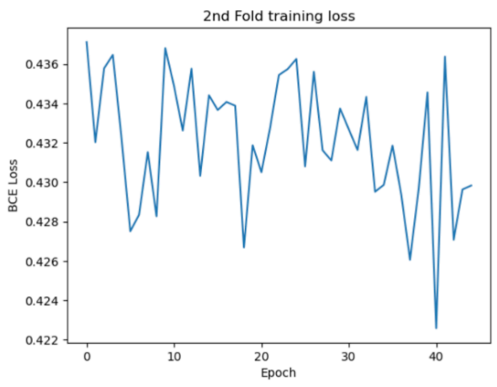


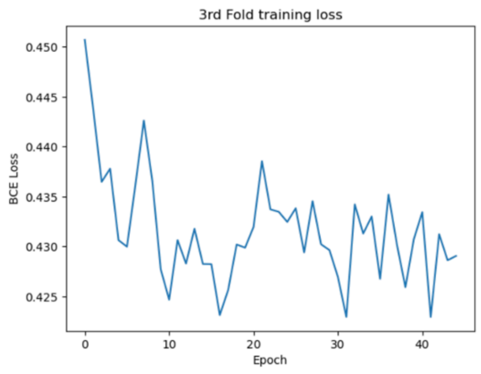

1st Fold AUROC - Epoch
AUROC
Epoch
training
validation


2nd Fold AUROC - Epoch
AUROC
Epoch
training
validation

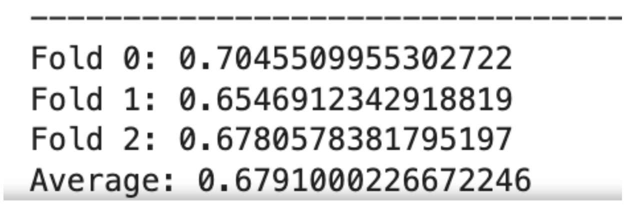

3rd Fold AUROC - Epoch
AUROC
Epoch
training
validation



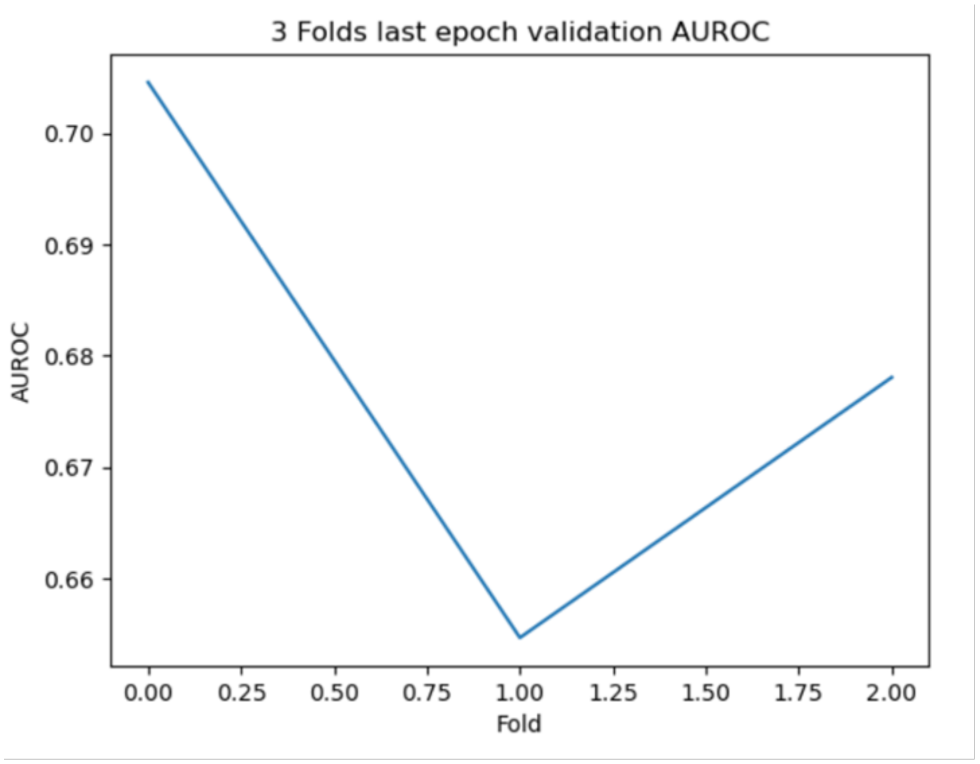


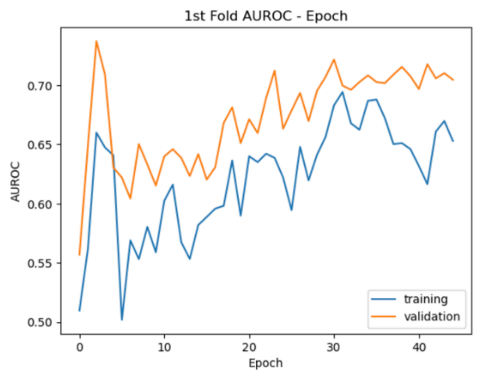


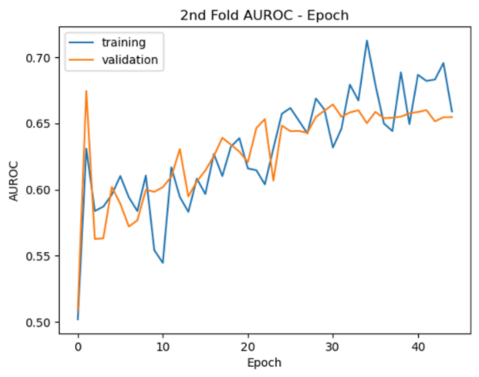
Results for 3.2.2

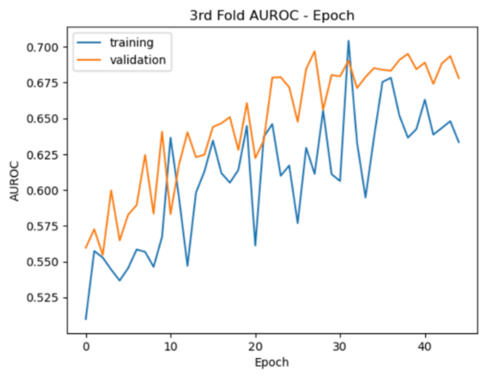


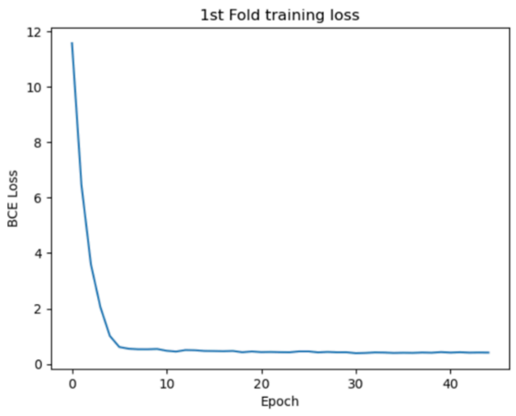
```
--------------------------------
Fold 0: 0.7045509955302722
Fold 1: 0.6546912342918819
Fold 2: 0.6780578381795197
Average: 0.6791000226672246
```

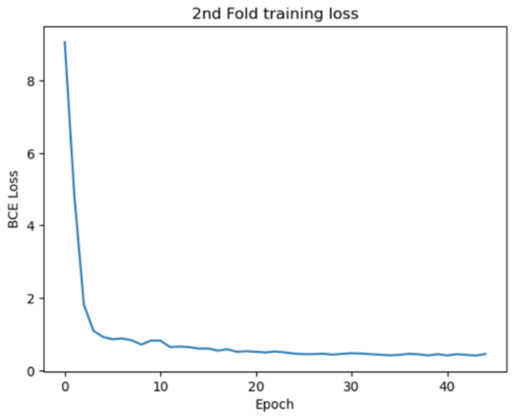

3 Folds last epoch validation AUROC
AUROC
Fold
0.70
0.69
0.68
0.67
0.66
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

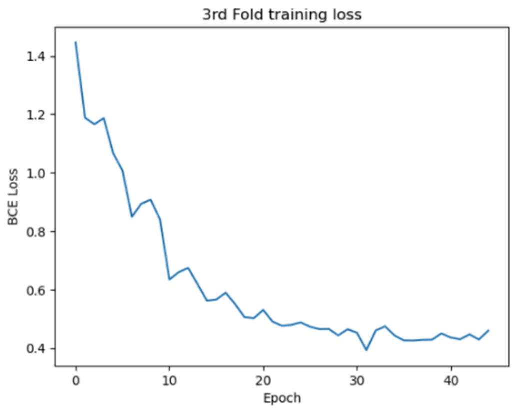

1st Fold AUROC - Epoch
AUROC
Epoch
training
validation


2nd Fold AUROC - Epoch
AUROC
Epoch
training
validation

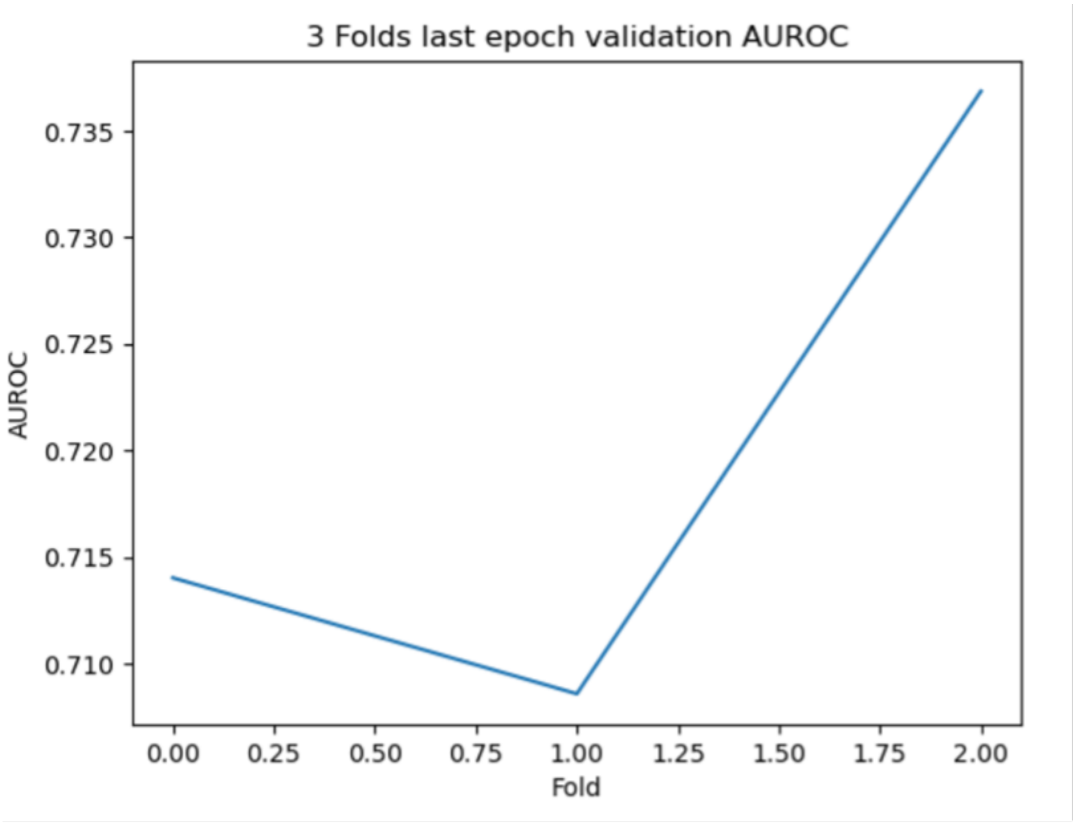

3rd Fold AUROC - Epoch
AUROC
Epoch
training
validation

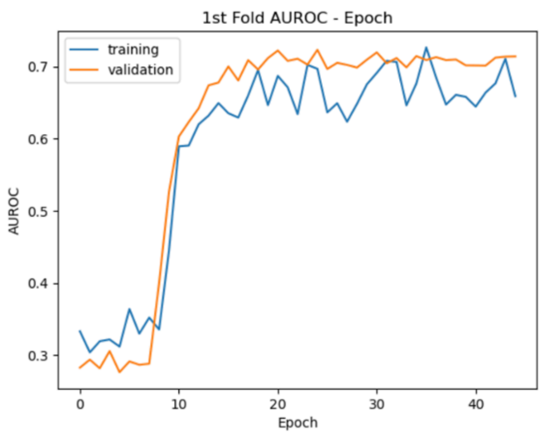

1st Fold training loss
BCE Loss
Epoch

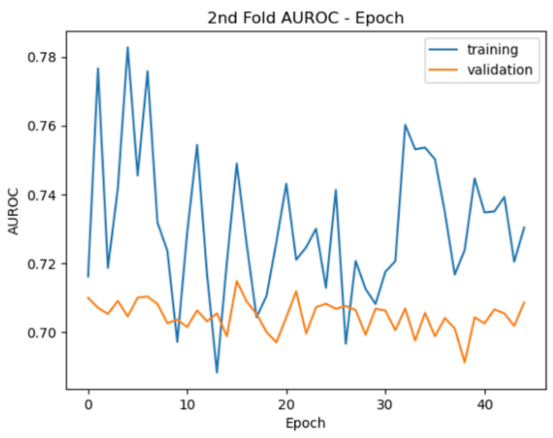

2nd Fold training loss
BCE Loss
Epoch

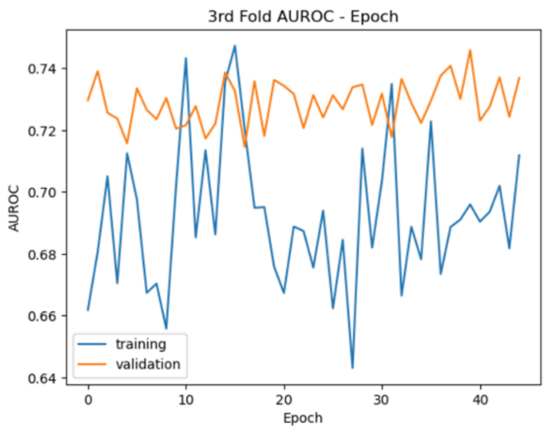

3rd Fold training loss
BCE Loss
Epoch

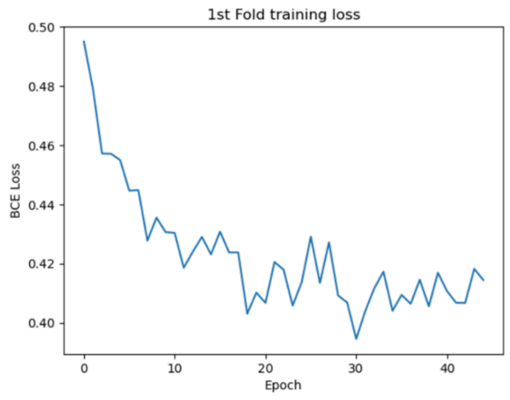
Results for 3.2.3

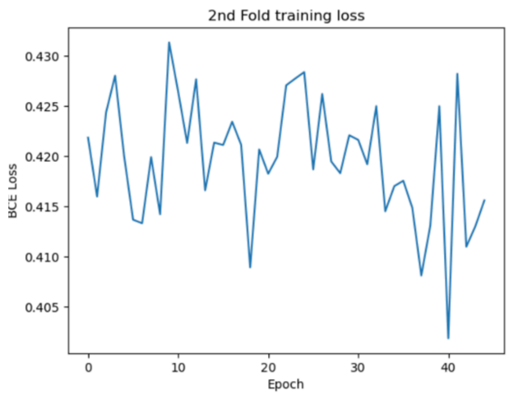


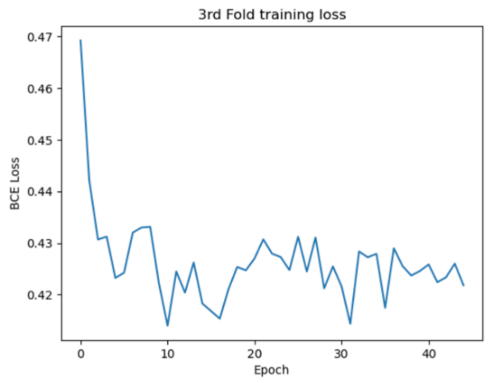

1st Fold AUROC - Epoch
training
validation
AUROC
Epoch
0.3
0.4
0.5
0.6
0.7
0
10
20
30
40


2nd Fold AUROC - Epoch
training
validation
AUROC
Epoch
0.70
0.72
0.74
0.76
0.78
0
10
20
30
40


3rd Fold AUROC - Epoch
training
validation
AUROC
Epoch
0.64
0.66
0.68
0.70
0.72
0.74
0
10
20
30
40







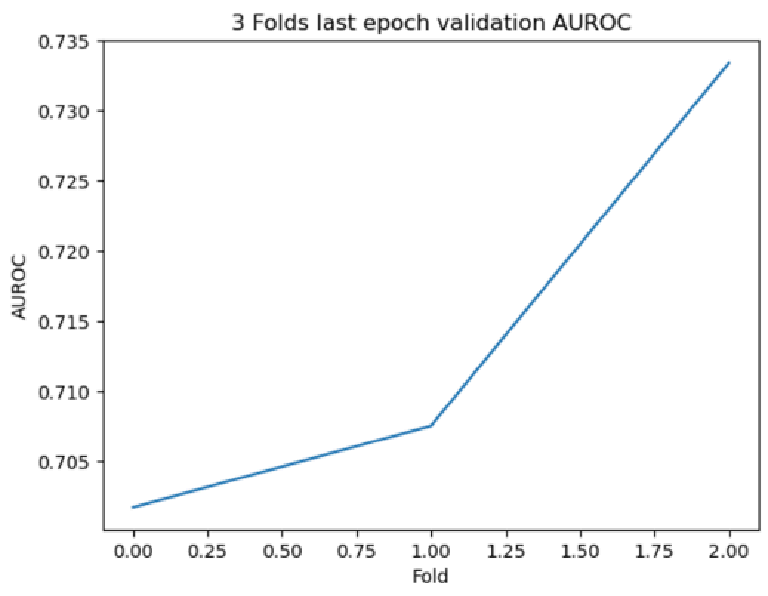
Code for 3.3

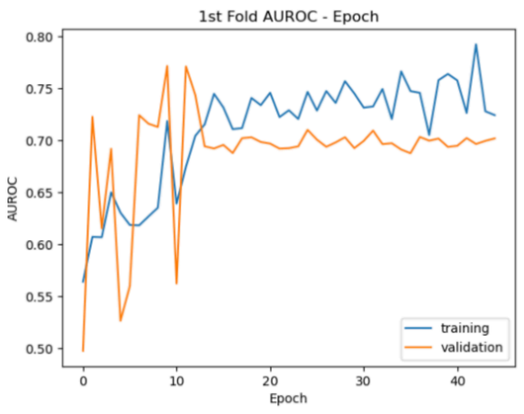
```
#Model: Multi-attention

# class Head(nn.Module):
#     """ one head of self-attention """
#
#     def __init__(self, head_size):
#         super().__init__()
#         self.Wk = nn.Linear(n_embd, head_size, bias=False)
#         self.Wq = nn.Linear(n_embd, head_size, bias=False)
#         self.Wv = nn.Linear(n_embd, head_size, bias=False)
#
#     def forward(self, x):
#         # input of size (batch, time-step, channels)
#         # output of size (batch, time-step, head size)
#         B, T, C = x.shape
#         k = self.Wk(x)  # (B,T,hs)
#         q = self.Wq(x)  # (B,T,hs)
#         # compute attention weights
#         alpha = q @ k.transpose(-2, -1) * k.shape[-1] ** -0.5  # (B, T, hs) @ (B, hs, T)
#         alpha = F.softmax(alpha, dim=-1)  # (B, T, T)
#         # perform the weighted aggregation of the values
#         v = self.Wv(x)  # (B,T,hs)
#         out = alpha @ v  # (B, T, T) @ (B, T, hs) -> (B, T, hs)
#         return out
#
#
# class MultiHeadAttention(nn.Module):
#     """ multiple heads of self-attention in parallel """
#
#     def __init__(self, num_heads, head_size):
#         super().__init__()
#         self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
#
#     def forward(self, x):
#         out = torch.cat([h(x) for h in self.heads], dim=-1)
#         return out
#
#
# class FeedFoward(nn.Module):
#     """ a simple linear layer followed by a non-linearity """
#
#     def __init__(self, n_embd):
#         super().__init__()
#         self.net = nn.Sequential(
#             nn.Linear(n_embd, 4 * n_embd),
#             nn.ReLU(),
#             nn.Linear(4 * n_embd, n_embd),
#         )
#
#     def forward(self, x):
#         return self.net(x)
#
#
# class Block(nn.Module):
#     """ Transformer block: communication followed by computation """
#
```

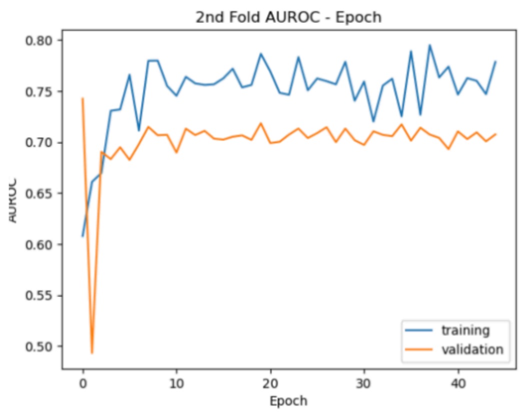
```
#     def __init__(self, n_embd, n_head):
#         # n_embd: embedding dimension =32, n_head: the number of heads we'd like
#         super().__init__()
#         head_size = n_embd // n_head
#         self.sa = MultiHeadAttention(n_head, head_size)
#         self.ffwd = FeedFoward(n_embd)
#         self.ln1 = nn.LayerNorm(n_embd)
#         self.ln2 = nn.LayerNorm(n_embd)
#
#     def forward(self, x):
#         x = x + self.sa(self.ln1(x))
#         x = x + self.ffwd(self.ln2(x))
#         return x
#
#
# class TissueGPT(nn.Module):
#
#     def __init__(self):
#         super().__init__()
#         # Project the 2 dimension spatial feature to 32 dimension
#         self.embedspatial=nn.Sequential(
#         nn.Linear(2, 32),
#         nn.LayerNorm(32))
#         self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_laye
#         self.ln_f = nn.LayerNorm(n_embd)  # final layer norm
#         self.lm_head = nn.Linear(n_embd, 1)
#         self.apply(self._init_weights)
#
#     def _init_weights(self, module):
#         if isinstance(module, nn.Linear):
#             torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
#             if module.bias is not None:
#                 torch.nn.init.zeros_(module.bias)
#         elif isinstance(module, nn.Embedding):
#             torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
#
#     def forward(self, x, targets=None):
#
#         B = x.size()[0]
#         T = x.size()[1]
#
```

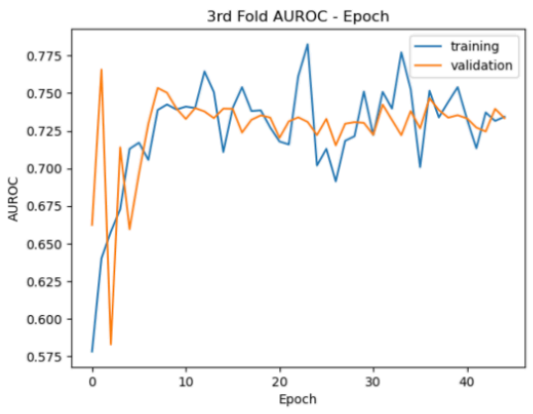
```
#         #!! With embedded Spatial information
#         # x= self.embedspatial(x[:,:,:2])+x[:,:,2:]
#
#         #!! With concat spatial information, n_embed =34
#         x = x
#
#         #!! Without spatial information
#         # x=x[:,:,2:]
#
#
#
#         x = self.blocks(x)  # (B,T,C)
#         x = self.ln_f(x)  # (B,T,C)
#         x = torch.sum(x, dim=1, keepdim=True)  # (B,1,C)
#         output = self.lm_head(x)  # (B,1,1)
#
#         B, T, C = output.shape
#         output = output.view(B, T * C)
#
#
#         return output
```

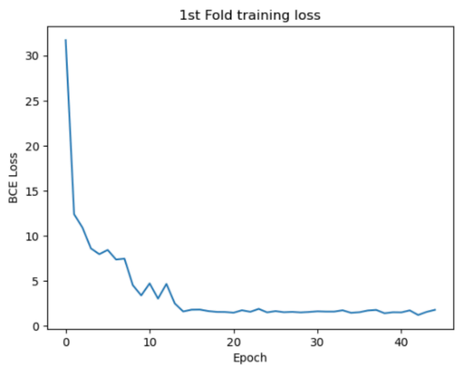
Results for 3.3.1

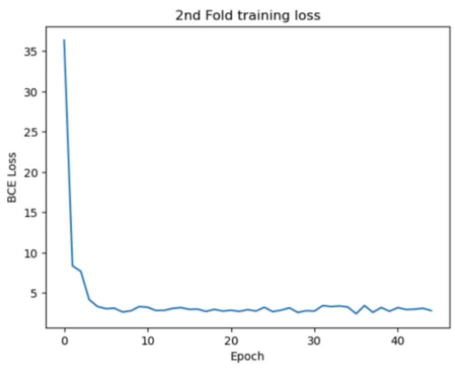


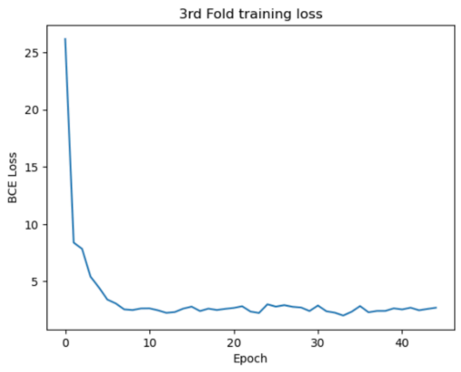

1st Fold AUROC - Epoch
AUROC
Epoch
training
validation


2nd Fold AUROC - Epoch
AUROC
Epoch
training
validation


3rd Fold AUROC - Epoch
AUROC
Epoch
training
validation

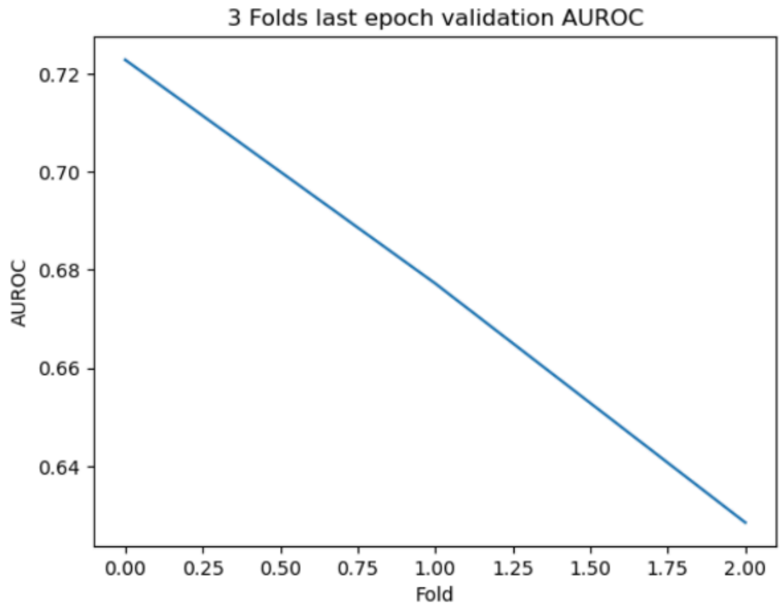

1st Fold training loss
BCE Loss
Epoch

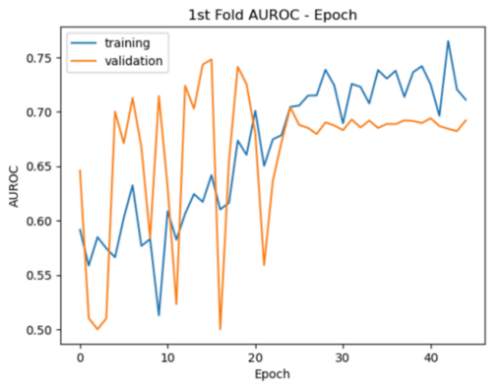
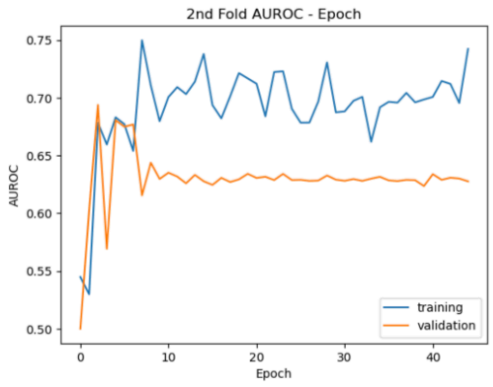
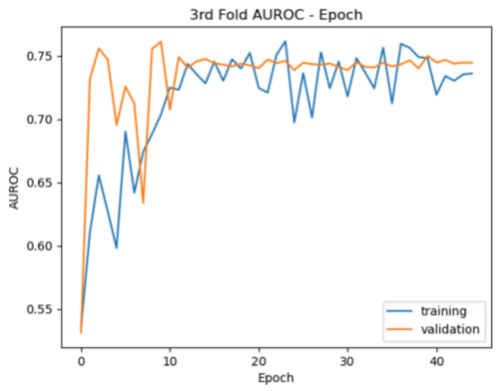

2nd Fold training loss
BCE Loss
Epoch

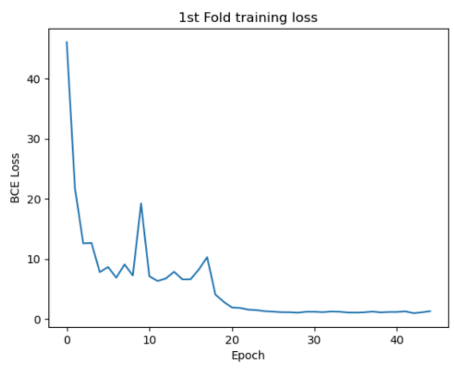

3rd Fold training loss
BCE Loss
Epoch

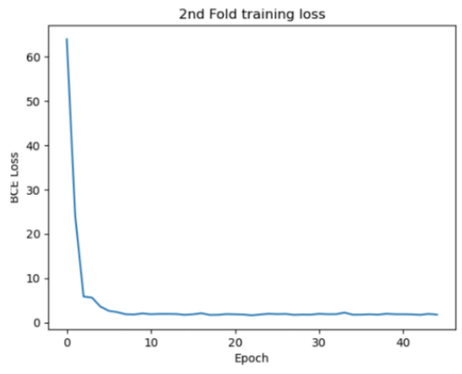
Results for 3.3.2

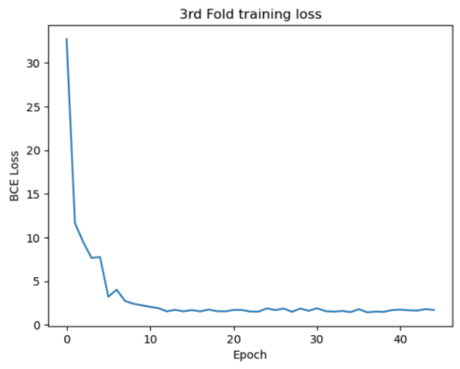
```
Average AUROC: 0.6761911953795444
```




1st Fold AUROC - Epoch
training
validation
AUROC
Epoch

2nd Fold AUROC - Epoch
training
validation
AUROC
Epoch

3rd Fold AUROC - Epoch
training
validation
AUROC
Epoch

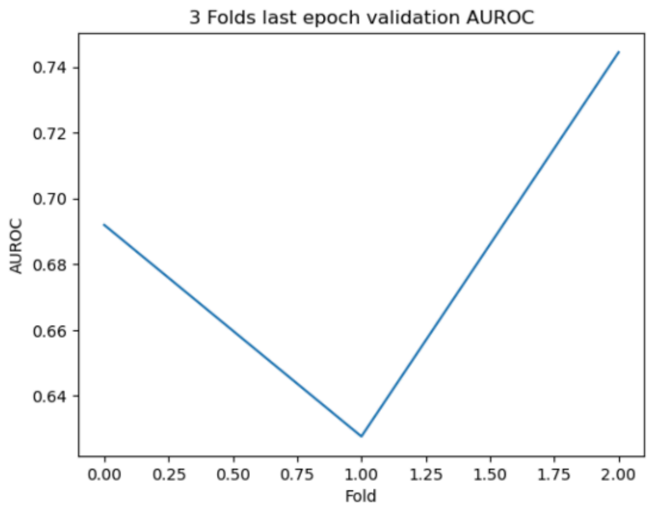


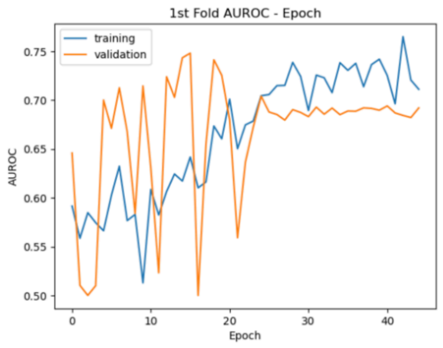


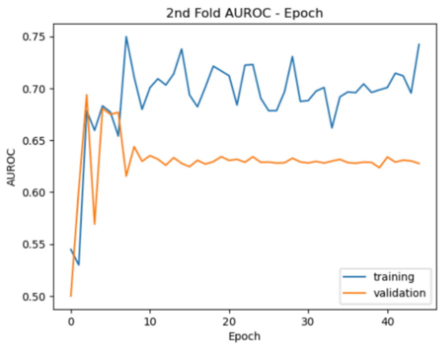


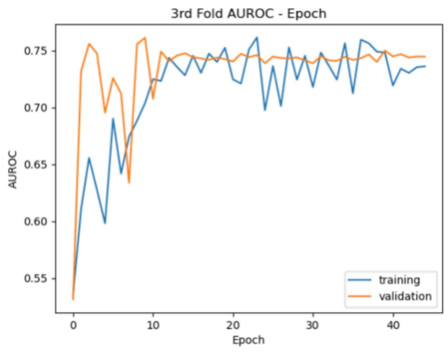
Results for 3.3.3

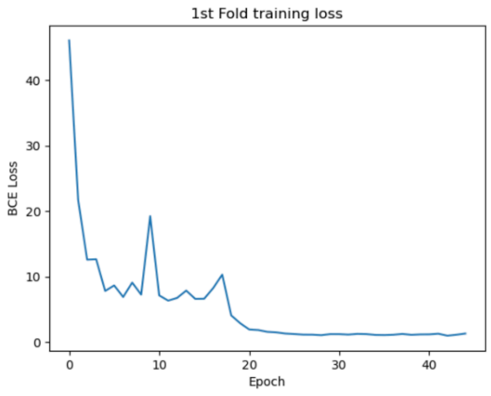
```
Average AUROC: 0.6880046396924439
```

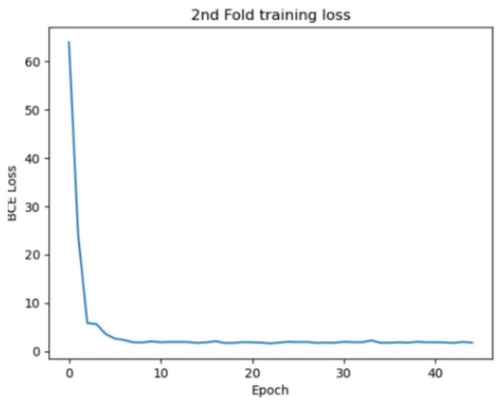


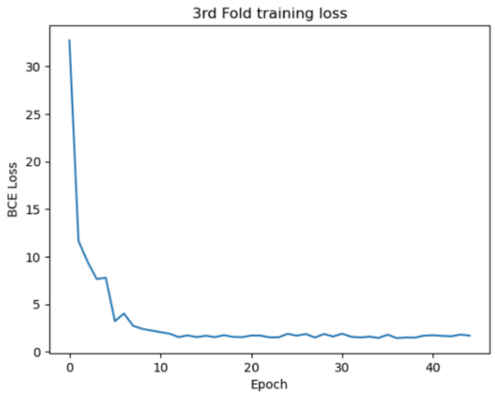

1st Fold AUROC - Epoch
training
validation
AUROC
Epoch


2nd Fold AUROC - Epoch
training
validation
AUROC
Epoch


3rd Fold AUROC - Epoch
training
validation
AUROC
Epoch


1st Fold training loss
BCE Loss
Epoch


2nd Fold training loss
BCE Loss
Epoch


3rd Fold training loss
BCE Loss
Epoch