Project A: Visual Interpretation of Convolutional Neural Networks

***Abstract*—This paper is a report for Project A of ECE1512 2022W, University of Toronto. In the report paper, we described four assigned tasks in detail, including CNN construction and training, statistic assessment, XAI based interpretation and Quantitative evaluation. For the attribution methods, we choose Grad-CAM, LIME and Ablation-CAM as a group of two people. We reviewed their paper and implemented corresponding XAI functions. Moreover, we tried to analyze the performance of the XAI methods based on quantitative evaluation and gave our explanations towards the experiment. This project remote repository is attached on GitHub: https://github.com/Awesome-guys-in-ECE1747/ECE1512_2022W_ProjectRepo_J.Xu_and_W.Xu**

I. Task 1: 1-Dimensional digit classification

In this task, we basically train an one-hot 1-D CNN model following typical procedure, including network construction, training and evaluation.

A. Question 1

In this section, we build a one-hot Conv-Net, including three convolutional layer, one flatten layer and one dense layer. The network structure is plotted by $model.summary()$ and shown as Fig. 1.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 40, 25)	150
conv1d_1 (Conv1D)	(None, 40, 25)	1900
conv1d_2 (Conv1D)	(None, 40, 25)	1900
flatten (Flatten)	(None, 1000)	0
dense (Dense)	(None, 10)	10010

Total params: 13,960
Trainable params: 13,960
Non-trainable params: 0

Fig. 1. Conv-Net Model

This is not a complex network, since there are only three sequential convolution layer for feature extracting and no additional structures. The implementation code is as followed.

```
weight_decay = 5e-4
model = Sequential()
#Your code starts from here
model.add(Input(shape=(40,1)))
model.add(Conv1D(25, kernel_size=5,
    padding='same', activation='relu',
    kernel_regularizer=regularizers.l2(
    weight_decay)))
model.add(Conv1D(25, kernel_size=3,
    padding='same', activation='relu',
    kernel_regularizer=regularizers.l2(
    weight_decay)))
model.add(Conv1D(25, kernel_size=3,
    padding='same', activation='relu',
    kernel_regularizer=regularizers.l2(
    weight_decay)))

model.add(Flatten())
model.add(Dense(10, activation='softmax',
    kernel_initializer=keras.initializers.
    RandomNormal(mean=0.0, stddev=0.5),
bias_initializer=keras.initializers.Zeros
    (), kernel_regularizer=regularizers.l2
    (weight_decay)))

model.summary()
```

B. Question 2

In this section, we applied the model constructed in the previous section to the MNIST1D dataset.

In this part, we use the TensorBoard to record the training procedure, which we also found could be standalone execute as a TensorFlow based analyzing tool which is running on localhost with port 6006 by default.

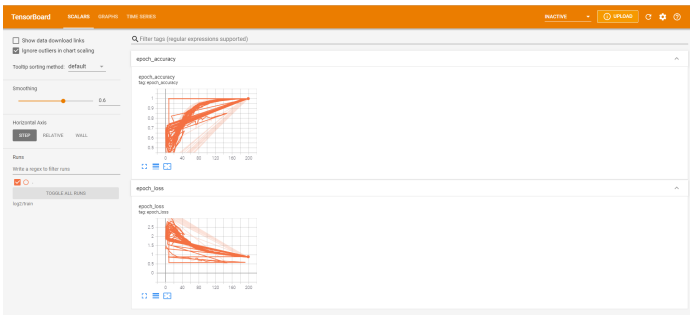


Fig. 2. TensorBoard GUI

As a standard procedure using TensorFlow, first of all, we compile this model and set the loss function to cross-entropy, the optimizer to Stochastic Gradient Descent(SGD) and the target metrics to accuracy as required.

Then we define a LearningRateScheduler, in order to control the learning rate during the training process. Learning rate is

similar to a ”step” of training process, which is typically set to 0.01 or 0.001. Learning rate only care about the magnitude, it somehow determine the precision of fitting parameters. In this part, we declare a dynamic learning rate to increase it speed to fit, while setting the magnitude to 0,01 according to the direction.

For the next step, we declare the object of TensorBoard, along with the EarlyStopping to accomplish a speed-up of training working with the dynamic learning rate. Then, in order

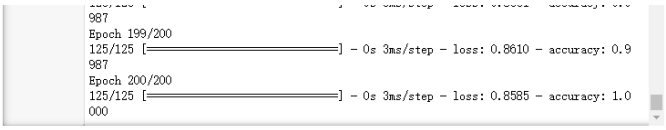


Fig. 3. Task1-Question2: Training Process Log

to be convenient for the following steps, we extract the dataset into several lists, and handle the data into correct form. Finally, we feed the data with the facilities into the model to execute training process and save the model as a static file afterwards.

A screen shot of the training log generated by TensorFlow is shown in Fig. 2.

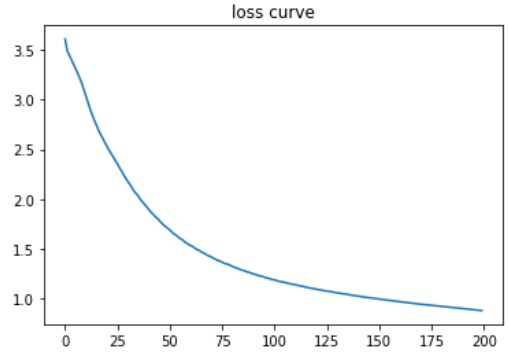
```
model.compile(loss=keras.losses.
    categorical_crossentropy,
optimizer=tensorflow.keras.optimizers.SGD
    (),
metrics=['accuracy'])

def lr_scheduler(epoch):
base_ep = 15
return 1e-3 * (.5 ** (epoch // base_ep))
lr_reduce_cb = keras.callbacks.
    LearningRateScheduler(lr_scheduler)
tensorboard_cb = keras.callbacks.
    TensorBoard(log_dir='log2',
    write_graph=True)
early_stopping_cb = keras.callbacks.
    EarlyStopping(patience=8, min_delta
    =0.)

# X = tensorflow.expand_dims(dataset['x'],
    axis=2)
train_x=dataset['x']
train_y=dataset['y']
train_x=train_x.reshape(4000,40,1)
train_y=tensorflow.keras.utils.
    to_categorical(train_y, num_classes
    =10)

# print(X.shape)
history=model.fit(x=train_x,y=train_y,
    epochs=200,
#                 steps_per_epoch=len(X) //
     32,
callbacks=[tensorboard_cb],
shuffle = True,
verbose=1)
model.save('MNIST1D.h5')
```

C. Question 3

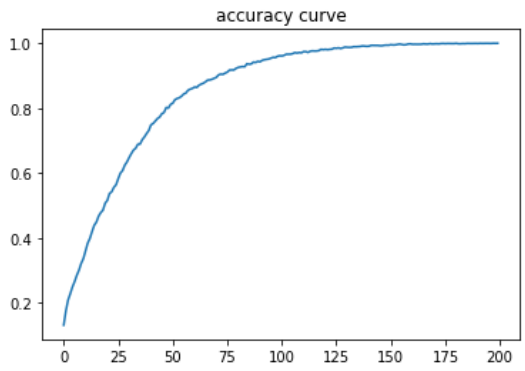
In this section, we finish the typical model evaluation by extracting several significant statistics. As well-known and commonly used metrics, those statistics all have easy-to-run function to achieve, and most of them are from $scikit-learn$.

1) SubQuestion a: We firstly plot the loss curve and accuracy curve of the training process. As implementation, we extract the data using the training history, which is a return value of $model.fit()$. This is a modification of the original notebook since the $model.fit\_generator()$ is deprecated in TensorFlow 2.7.0 which is the version we used for this project, and it no longer saves training statistics of accuracy and loss during the training process in the log, which could be extracted by $tf\_record.tf\_record_iterator()$ in the past.

As a matter of fact, this plot is generated only when we execute the training process (i.e. $model.fit()$ function). And it is shown in Fig. 3.



(a) Loss



(b) Accuracy

Fig. 4. Loss and Accuracy Curve Plots on MNIST-1D dataset

The code is quite simple as shown below, whose history object is define in the previous section as mentioned.

```
train_acc = history.history['accuracy']
train_loss = history.history['loss']
plt.plot(train_acc)
plt.figure()
plt.plot(train_loss)
```

2) SubQuestion b: In this subquestion, our goal is to test the overall classification accuracy on the test set. As implementation, we first import test sets to x_test and y_test respectively. Then, we reshape the data in the x_test. Next, we exploit $model.predict()$ to obtain the predicted output y_pred. Finally, we compare the predicted values with the data in y_test to calculate the overall classification accuracy.

```
Overall classification accuracy for all classes:0.919
```

Fig. 5. Task1-Question3b: Overall accuracy

As a result, the overall accuracy is 0.919, which conforms to our expectation. Figure 4 shows the result of our experiment.

The code used in this subquestion is showed below.

```
# Use Scikit-learn to calculate stats
from sklearn.metrics import accuracy_score
    , precision_score, recall_score,
    f1_score
from sklearn.metrics import
    classification_report

# Q3.b get the prediction from the test
    set
x_test = dataset['x_test']
y_test = dataset['y_test']
x_test=x_test.reshape(1000,40,1)
# predict_classes removed in tf 2.6.0
y_pred = model.predict(x_test)
y_predicted = np.argmax(y_pred,axis=1)
print('Overall classification accuracy for
     all classes:'+str(np.sum(y_predicted
    ==y_test)/y_test.shape[0]))
```

3) SubQuestion c: In this subquestion, our goal is to test the classification accuracy of each class. For class 0 to class 9, we compute the accuracy respectively. Our code of this subquestion is showed below. We make use of the results of $y\_predicted$ in subquestion b. For each class, if $y\_predicted$ == y_test, then we increment res, which is counter to record the number of samples that are classified correctly. Later, we can simply compute the accuracy by dividing res by $true\_i.shape[0]$, which is the total sample number of class i.

The code used in this subquestion is showed below.

```
# Q3.c class-wise classification
for i in range(10):
true_i=np.where(y_test==i)[0]
res=0
for j in true_i:
if y_predicted[j]==i:
res=res+1
print('class '+str(i)+' accuracy:'+str(res
    /true_i.shape[0]))
```

We find the lines which belongs to class i and find the ones which is correct in these lines to calculate the accuracy. The results of subquestion b is showed in figure 5.

```
class 0 accuracy:0.9803921568627451
class 1 accuracy:0.8557692307692307
class 2 accuracy:0.898876404494382
class 3 accuracy:0.9811320754716981
class 4 accuracy:0.8962264150943396
class 5 accuracy:0.9183673469387755
class 6 accuracy:0.9595959595959596
class 7 accuracy:0.8958333333333334
class 8 accuracy:0.9795918367346939
class 9 accuracy:0.8235294117647058
```

Fig. 6. Task1-Question3b: class-wise accuracy

As a result, class 0, class 3, class 5, class 6, class 8 have relatively higher accuracy, which are all greater than 90%, while class 1, class 2, class 4, class 7 and class 9 have relatively lower accuracy, which are all less than 90%.

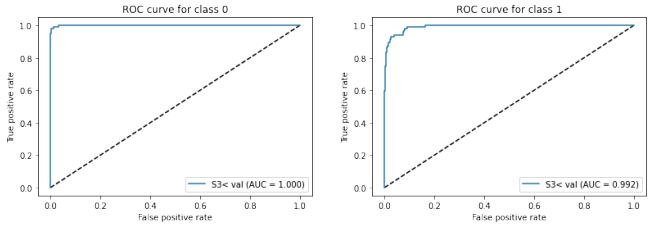
4) SubQuestion d: In this subquestion, our goal is to plot the ROC and AUC curves for each class. First of all, we transform the test set and reshape it to a 2D set. For each class, we compute the parameters FPR, TPR by calling the function roc_curve. Then we input the FPR and TPR parameters into function $auc()$, and obtained the returned value AUC, which is used to plot the graph.

The code used in this subquestion is showed below.

```
# Q3.d ROC and AUC curve for every class
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
# print(dataset['y_test'].shape)
real_y=np.zeros((dataset['y_test'].size
    ,10))
for i in range(y_test.size):
real_y[i,y_test[i]]=1
for i in range(10):
FPR, TPR, thresholds_keras = roc_curve(
    real_y[:,i], y_pred[:,i])
AUC = auc(FPR, TPR)
print("AUC : ", AUC)
plt.figure()
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(FPR, TPR, label='S3< val (AUC =
    {:.3f})'.format(AUC))
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve for class '+str(i))
plt.legend(loc='best')
plt.show()
```

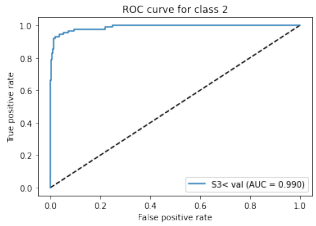
The AUC and ROC curves we plotted by executing the code are listed in figure 6.

5) SubQuestion e: In this sebquestion, our goal is to plot the normalized confusion matrix. First, we will get the confusion matrix cm using $confusion\_matrix$ function. We use y_test which represents the real class labels of test set and $y_predicted$ which represents the predicted labels using the model. We transformed them as $list$ to fit into the function. Then we use $cm.astype('float')/cm.sum(axis = 1)[:, np.newaxis]$ to normalize the confusion matrix. After

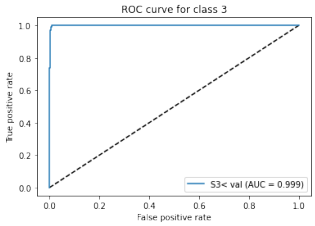


(a) Class 0

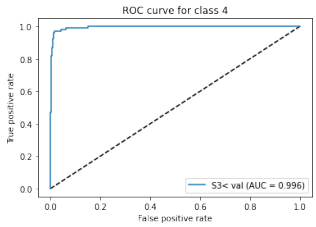
(b) Class 1



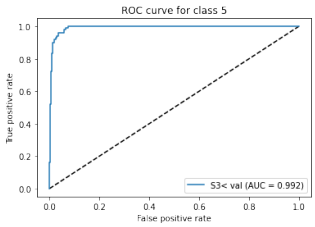
(c) Class 2



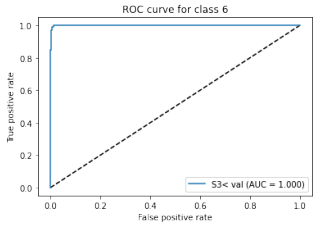
(d) Class 3



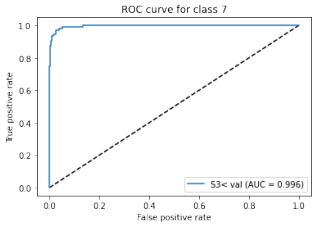
(e) Class 4



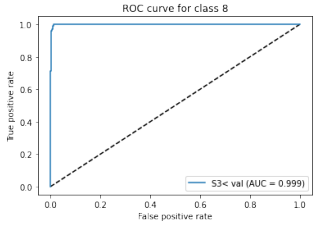
(f) Class 5



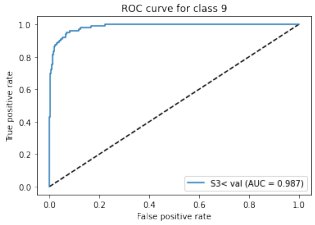
(g) Class 6



(h) Class 7



(i) Class 8



(j) Class 9

Fig. 7. Class-wise ROC Curve Plots on MNIST-1D dataset

that, we will set the plot cmap to blue confusion matrix plot, amd set the parameter in the plot ax. Finally, we will fit all the data in normalized confusion matrix into the data. The code used in this subquestion is showed below, and the result plot is displayed in figure 7.

```
from sklearn.metrics import
    confusion_matrix
cm=confusion_matrix(list(y_test),list(
    y_predicted))
cm_normalized = cm.astype('float') / cm.
    sum(axis=1)[:, np.newaxis]
fig, ax = plt.subplots()
im = ax.imshow(cm, interpolation='nearest'
    , cmap=plt.cm.Blues)
ax.figure.colorbar(im, ax=ax)
classes=[0,1,2,3,4,5,6,7,8,9]
# print(classes)
# We want to show all ticks...
ax.set(xticks=np.arange(cm_normalized.
    shape[1]),
yticks=np.arange(cm_normalized.shape[0]),
# ... and label them with the respective
    list entries
xticklabels=classes, yticklabels=classes,
ylabel='True label',
xlabel='Predicted label')
plt.setp(ax.get_xticklabels(), ha="right",
rotation_mode="anchor")
thresh = cm_normalized.max() / 2.
for i in range(cm_normalized.shape[0]):
for j in range(cm_normalized.shape[1]):
ax.text(j, i, format(cm_normalized[i, j],
    '.2f'),
ha="center", va="center",
color="white" if cm_normalized[i, j] >
    thresh else "black")
fig.tight_layout()
```

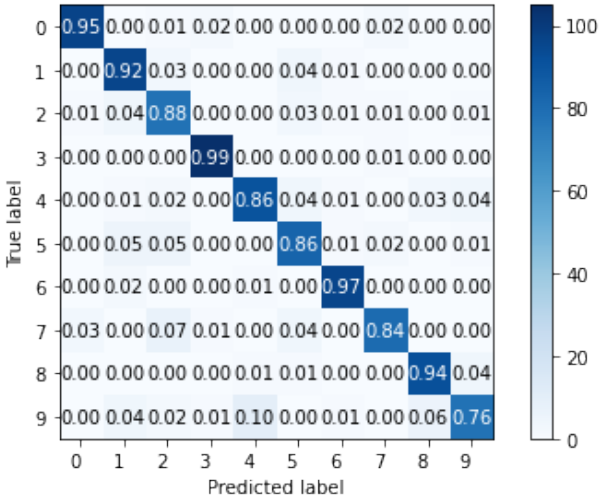


Fig. 8. Normalized Confusion Matrix Plot of MNIST-1D model

6) SubQuestion f: In this subquestion, our goal is to calculate Precision, Recall, and F-1 score on the test set. We call $classification\_report()$ to compute the statistics.

The code used in this subquestion is showed below.

```
# Q3.f
print(classification_report(y_true=y_test,
    y_pred=y_predicted))
```

The result of the calculation is displayed in figure 8. As a result, the accuracy on total test set if 0.92, the f1-score is 0.92 and the recall is 0.92.

	precision	recall	f1-score	support
0	0.97	0.98	0.98	102
1	0.92	0.86	0.89	104
2	0.87	0.90	0.88	89
3	0.96	0.98	0.97	106
4	0.91	0.90	0.90	106
5	0.83	0.92	0.87	98
6	0.97	0.96	0.96	99
7	0.96	0.90	0.92	96
8	0.91	0.98	0.95	98
9	0.89	0.82	0.86	102
accuracy			0.92	1000
macro avg	0.92	0.92	0.92	1000
weighted avg	0.92	0.92	0.92	1000

Fig. 9. Precision, Recall and F-1 score

D. Question 4

To do this question, we did some coding in our program, the codes are listed below:

Success Examples:

```
index 10 true class: 0 predicted class: 0
index 412 true class: 1 predicted class: 1
index 729 true class: 2 predicted class: 2
```

Failure Examples:

```
index 236 true class: 0 predicted class: 7
index 6 true class: 1 predicted class: 2
index 25 true class: 1 predicted class: 4
```

All the concrete Success/Failure cases are listed in 'MNIST1D.ipynb', if you want to refer to more examples. The most misclassification lies in class 1 and 9. It is obvious class-wise accuracy. There are several reasons for this. We think the most important reason is the unbalanced dataset. We can see from ROC curve. It is obvious that class 1 and class 9 has a worse ROC curve than other classes. The more AUC is closer to 1.0, the easier it will be to maintain a high precision and recall at the same time. And see from our results, we can clearly see from the ROC curve, the class 1 and 9 have the lowest AUC values. This is also shown in the $Question3-f$, from the precision and recall table, we can see that the recall and precision is relatively lower than that of other classes. With low recall rate, the model cannot detect this class very well, and with a low precision, it's hard to trust its correctness when predicting this class. Class 9 has a lowest performance considering both precision and recall, so it has the lowest accuracy.To solve this, we can use undersampling, oversampling or generate synthetic data.

The second reason is the overfitting on the training set. It's likely that if the design is too complex compared to the relatively simple dataset, it'll lead to overfitting. The training accuracy may be very high. However, the testing set will not get a high precision due to overfitting.

II. Task 2: CNN interprectation

This section introduces our interpretation of 1-D CNN model based on MNIST-1D dataset using 3 different attribution methods, including our literature review, discussion and implementation of the XAI attribute algorithms.

A. Grad-CAM

In previous studies, researchers introduced CAM to explain the CNN. However, CAM requires to modify the structure of original training models. It limits the usage of CAM greatly, because the cost of retraining the model is quite high for the published model. It is almost impossible to retrain them. To solve this problem, Grad-CAM was proposed, which is similar to CAM. It also gets the weight of every feature maps and calculate the results accordingly. The difference lies in the calculation of weights. CAM replaced full-connection layer with GAP layer and retrain this whole model. In contrast to that, Grad-CAM put forward a new way to do it. It calculates the weight by averaging the gradients which is equivalent to the CAM. It makes CAM applicable to all the existing models.

The core algorithm of Grad-CAM is based on CAM which proposed: class c gets the final classification score Y_c with a linear combination of its global average pooled last convolutional layer feature maps A_k.

$$Y^c = \Sigma_k w_k^c \Sigma_i \Sigma_j A_{ij}^k \tag{1}$$

So, first, it defines the weight of feature map k to class c as α_k^c, and it calculates the weight using the global gradients2:

$$\alpha_k^c = \frac{1}{Z} * \Sigma_i \Sigma_j \frac{\partial y^c}{\partial A_{ij}^k} \tag{2}$$

In which, y_c is the gradient of the score for class c, A_k is the feature map activation. $\frac{\partial y^c}{\partial A_{ij}^k}$ is the gradient achieved by backpropagating. $\frac{1}{Z}$ is global averaging. After calculating the α_k^c, the algorithm computed the weighted combination of forward activation maps, followed by a ReLU function:

$$L_{Grad-CAM}^c = ReLU(\Sigma_k \alpha_k^c A^k)$$

The advantages for Grad-CAM is quite obvious: First, compared with CAM, it does not need to change the model and retrain it which costs a lot of time and money. The novelty is the usage of propagation and gradients to compute the weights of feature maps.It makes the visualization of published models possible. Second is that the algorithm is quite easy to code and maintain.

The disadvantages for Grad-CAM is that: it considers the global features for all classes. As a result, Grad-CAM cannot localize the targets when faced with many targets of the same class. If there are many objects of the same class in an image, it cannot localize the targets well or it can only locate part of them.

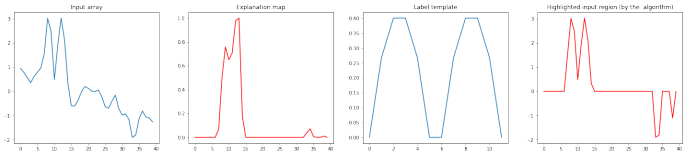


Fig. 10. Grad-CAM 1-D XAI plots

B. LIME

LIME, Local Interpretable Model-agnostic Explanations, is an algorithm that can explain the predictions of any classifier or regressor in a faithful way by approximating it locally with an interpretable model. According to the M.T.Ribeiro et al., the trustness of models consists of three parts, trustness of prediction, trustness of model and that people could evaluate model as a whole so that they could evaluate the exact impact of different components. The paper indeedly introduce 3 parts of methods corresponding to the three part of trustness, and LIME, is what they used to evaluate the prediction, which is a XAI method.

There are four principles for LIME to assure, first, an essential criterion for explanations is that the model have to be interpretable, besides, another criterion is local fidelity, as it must be at least locally faithful for an explanation to be meaningful, moreover, the explainer should be model-agnostic, i.e. it could treat the original model as a black box and be able to explain any model which fit the criterions. Last but not the least, in addition to explaining prediction, the explainer also provides a global perspective, which is important to ascertain trust in the model.

```
def perturb_1d(input, perturb, sec_size):
   mask = np.zeros(input.shape)[0]
   not_masked = np.where(perturb == 1)[0]
   for i, val in enumerate(not_masked):
      if val != 0:
         for j in range(sec_size):
            mask[sec_size * i + j] = 1
   perturbed = input * mask[:, np.newaxis]
   return perturbed

def LIME_1d(input,
         model,
         label,
         num_perturb=300,
         sec_size=4,
         kernel_w=0.25,
         num_feats=4):

   num_superpixels = math.ceil(input.shape[1]
       / sec_size)
   perturbs = np.random.binomial(1, 0.5, size
       =(num_perturb, num_superpixels))
   preds = []
   for i in perturbs:
      perturbed = perturb_1d(input, i,
          sec_size)
      pred = model.predict(perturbed)
      preds.append(pred)
   preds = np.array(preds)
   orig_img = np.ones(num_superpixels)[np.
       newaxis, :]
   dists = sklearn.metrics.pairwise_distances(
       perturbs,
                                       orig_img,
                                       metric='
                                           cosine'
                                           ).ravel
                                           ()
   weights = np.sqrt(np.exp(-(dists**2) /
       kernel_w**2))
   lime_model = LinearRegression()
   lime_model.fit(perturbs, preds[:, :, label
       ], weights)
   c = lime_model.coef_[0]
   top_feats = np.argsort(c)[-num_feats:]
   mask_superpixel = np.zeros(num_superpixels)
   mask_superpixel[top_feats] = True
   mask = np.zeros(input.shape[1])
   for i, val in enumerate(mask_superpixel):
      if val != 0:
         for j in range(sec_size):
            mask[sec_size * i + j] = 1
   perturbed = input * mask[:, np.newaxis]

   explanation = np.zeros(input.shape[1])
   for i, c_i in enumerate(c):
      for j in range(sec_size):
         if c_i > 0:
            explanation[sec_size * i + j] =
                c_i

   return explanation, perturbed[0]

def perturb_2d(input, perturb, segs):
   mask = np.zeros(segs.shape)
   not_masked = np.where(perturb == 1)[0]
   for i in not_masked:
      mask[segs == i] = 1
   perturbed = copy.deepcopy(input)
   perturbed = perturbed * mask[:, :, np.
       newaxis]
   return perturbed


def LIME_2d(input,
         model,
         label,
         num_perturb=300,
         kernel_size=4,
         max_dist=200,
         ratio=0.2,
         kernel_w=0.25,
         num_feats=10):
   superpixels = skimage.segmentation.
       quickshift(input,
                                       kernel_size
                                           =
                                           kernel_size
                                           ,
                                       max_dist=
                                           max_dist
                                           ,
                                       ratio=
                                           ratio
                                           )
   num_superpixels = np.unique(superpixels).
```

```
    shape[0]
perturbs = np.random.binomial(1, 0.5, size
    =(num_perturb, num_superpixels))
preds = []
for i in perturbs:
   perturbed = perturb_2d(input, i,
       superpixels)
   pred = model.predict(perturbed[np.
       newaxis, :, :, :])
   preds.append(pred)
preds = np.array(preds)
orig_img = np.ones(num_superpixels)[np.
    newaxis, :]
dists = sklearn.metrics.pairwise_distances(
    perturbs,
                                orig_img,
                                metric='
                                    cosine'
                                ).ravel
                                ()
weights = np.sqrt(np.exp(-(dists**2) /
    kernel_w**2))
lime_model = LinearRegression()
lime_model.fit(perturbs, preds[:, :, label
    ], weights)
c = lime_model.coef_[0]
top_feats = np.argsort(c)[-num_feats:]
mask = np.zeros(num_superpixels)
mask[top_feats] = True
perturbed = perturb_2d(input, mask,
    superpixels)
return perturbed
```

C. Ablation-CAM

The Ablation-CAM creatively uses ablation analysis to determine the importance of individual feature map units for different classes. It proposes a novel "gradient-free" visualization approach which avoids use of gradients and at the same time, produce high quality class-discriminative localization maps.

The core algorithm of Ablation-CAM is not complex: it uses the value of slope to describe the effect of ablation of individual unit k by the following formula:

$$slope = \frac{y^c - y^c_k}{||A_k||}$$

In the formula, y^c stands for activation score of class c, which represent the entire class activation status. y^c_k indicates the value of the function for absence of unit k, where A_k is the baseline. Those concepts lead us to ablation study, which is the basic principle of the method.

Ablation study is a method to distribute the influencing importance of different factors by controlling the variable while switching the combination of potential factors, and also their standalone. For example, if we'd like to know whether A or B component of medicine could improve the effect of an old medicine C. We could compare $C + A$, $C + B$ and also $C+A+B$ with the baseline of C. We could know if the A or B or they together are able to improve the effect. In the instance of Ablation-CAM, different unit k is the "component", and the whole feature map is so-called baseline, A_k. Thus, using slope described in the previous formula could represent the importance of a single unit to the feature map.

However practically, norm $||A_k||$ is hard to compute due to its large size and hence the slope could be approximately presented by the following formula, assuming a very small value.

$$w^c_k = \frac{y^c - y^c_k}{y^c}$$

As the algorithm, Ablation-CAM can then be obtained as weighted linear combination of activation maps and corresponding weights from the formula above, which is somehow similar to that of Grad-CAM.

$$L^C_{Ablation-CAM} = ReLU(\sum_k w^c_k A_k)$$

There are a number of advantages and features of Ablation-CAM. Firstly, a significant contribution and novelty of the Ablation-CAM is the ablation analysis it used to decide the weights of feature map units. Secondly, it could produce a coarse localization map highlighting the regions in the image for prediction. Thirdly, compare to other CAM methods, this approach works essentially better when it is full connected to obtain the result, which is known as final linear classifier, and have as good performance as other gradient-based CAM methods when evaluating other CNNs. Last but not the least, the approach introduce a gradient-free principle which avoids use of gradient as Grad-CAM does and produce a high-quality class-wise localization maps, which helps it to adapt into any CNN based architecture.

However, the approach have some limitations as well. First of all, the computational time required to generate a single Ablation-CAM is much grater than the required for Grad-CAM, as it has to iterate over each feature map to ablate it and check the drop in class activation score correspondingly. On the hand, the Ablation-CAM only benefits the interpretation where last convolutional layer is not followed immediately by decision nodes, yet show the same performance statistically as other CAM methods.

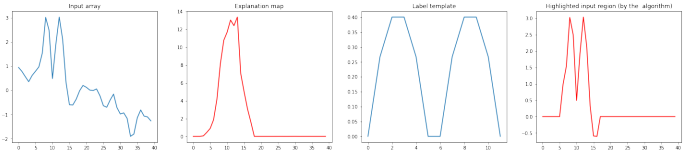


Fig. 11. Ablation-CAM 1-D XAI plots

The result plot and our implementation code is as followed.

```
def ablation_cam_1d(input_model, image,
    layer_name):

   output = input_model.output
   conv_layer_output = input_model.get_layer(
       layer_name).output
```

```
    ac_model = keras.models.Model([input_model.
        input],
                              [conv_layer_output,
                                  output])

    ff_results = ac_model([image])
    all_fmap_masks, predictions = ff_results
        [0], ff_results[-1]

    predicted_label = np.argmax(predictions,
        axis=1)[0]

    y_c = predictions[0][predicted_label]

    w_t = np.zeros(input_model.get_layer(
        layer_name).get_weights()[1].shape)

    all_weigths = input_model.get_layer(
        layer_name).get_weights().copy()
    zero_weigth = all_weigths[0][:, :, 0] * 0
    weigth_local = [np.zeros(all_weigths[0].
        shape)]
    weigth_local.append(np.zeros(all_weigths
        [1].shape))
    for i in range(w_t.shape[0]):
       weigth_local[0] = all_weigths[0].copy()
       weigth_local[0][:, :, i] = zero_weigth
       input_model.get_layer(layer_name).
           set_weights(weigth_local)
       y_k = input_model.predict([image])[0][
           predicted_label]
       w_t[i] = (y_c - y_k) / y_c

    a_k = all_fmap_masks[0]
    ab_map = a_k * w_t

    explanation = np.sum(ab_map, axis=1)
    return explanation


def ablation_cam_2d(input_model, image,
    layer_name):

    output = input_model.output
    conv_layer_output = input_model.get_layer(
        layer_name).output
    ac_model = keras.models.Model([input_model.
        input], [conv_layer_output, output])

    ff_results = ac_model([image])
    all_fmap_masks, predictions = ff_results
        [0], ff_results[-1]

    pred_label = np.argmax(predictions, axis=1)
        [0]

    y_c = predictions[0][pred_label]

    w_t = np.zeros(input_model.get_layer(
        layer_name).get_weights()[1].shape)

    all_weigths = input_model.get_layer(
        layer_name).get_weights().copy()
    zero_weigth = all_weigths[0][:, :, :, 0] *
        0
    weigth_local = [np.zeros(all_weigths[0].
        shape)]
    weigth_local.append(np.zeros(all_weigths
        [1].shape))
    for i in range(w_t.shape[0]):
       weigth_local[0] = all_weigths[0].copy()
       weigth_local[0][:, :, :, i] =
           zero_weigth
       input_model.get_layer(layer_name).
           set_weights(weigth_local)
       y_k = input_model.predict([image])[0][
           pred_label]
       w_t[i] = (y_c - y_k) / y_c

    a_k = all_fmap_masks[0]
    ab_map = a_k * w_t

    explanation = np.sum(ab_map, axis=2)
    explanation = np.maximum(explanation, 0)

    return explanation
```

III. Task 3: Biomedical image classification and interpretation

In this task, we basically apply the previous evaluation codes on HMT dataset with a 2-D CNN model following typical procedure, including plotting of test and attribution methods selected.

A. Question 1

In this section, we finish the typical model evaluation as we did in task 1 by extracting several significant statistics.As well-known and commonly used metrics, those statistics all have easy-to-run function to achieve, and most of them are from $scikit-learn$.

1) Overall classification accuracy: First of all, we implemented the overall classification accuracy evaluation. We load the model trained before, use the $test\_generator.classes$ function to get the test set for labels which is noted as y_test. Then, we use the test_generator to predict the results with this model, the result names y_pred. To achieve the label from the predict results, we then apply $np.argmax$ function to y_pred in every row and get the predicted label $y\_predicted$. So, for the overall classification accuracy, we need to compare the predicted labels with the true labels, calculate the number of pairs which have the same label.The function is realized by $np.sum(y\_predicted == y_test)$, and divide it with $y\_test.shape[0]$ which represents the amount of overall data, we will get the overall accuracy. Here is the code:

```
    from sklearn.metrics import accuracy_score,
        precision_score, recall_score,f1_score
    from sklearn.metrics import
       classification_report
    from sklearn.metrics import roc_curve, auc
    from sklearn.metrics import
       confusion_matrix

    test_generator.reset()
    y_test=test_generator.classes
    y_pred=model.predict(test_generator)
```

```
y_predicted=np.argmax(y_pred, axis=1)
print('Overall classification accuracy for
    all classes:'+str(np.sum(y_predicted==
    y_test)/y_test.shape[0]))
```

The result is 83.06%,which conforms to our expectation. The result is shown in Fig12:

```
Overall classification accuracy for all classes:0.8306451612903226
```

Fig. 12. Overall classification accuracy

2) class-wise classification accuracy: In this part, we aim to discuss the classification accuracy for every class using the test set. For class 0 to 7, we use similar methods as the work in overall classification accuracy. We uses the results of y_test in the previous code which represents the real class of the test set. For every class, we firstly select from the test set which has the same label as current index i. We uses $np.where$ function to get this. Then we will calculate the number of predicted labels in the index which hit the right labels. We use res as a counter to do that. For each class, if $y\_predicted == y\_test$, then we increment res. Later, we can simply compute the accuracy by dividing res by $true\_i.shape[0]$,which is the total sample number of class i. Here is the code:

```
for i in range(8):
true_i=np.where(y_test==i)[0]
res=0
for j in true_i:
   if y_predicted[j]==i:
      res=res+1
print('class '+str(i)+' accuracy:'+str(res/
    true_i.shape[0]))
```

As a result, class 0, class 1, class 5, class 6, class 7 have relatively higher accuracy, which are all greater than 90%, while class 2, class 3, class 4 have relatively lower accuracy, which are all less than 90%. The results are shown in Fig13:

```
class 0 accuracy:0.9193548387096774
class 1 accuracy:0.8709677419354839
class 2 accuracy:0.5967741935483871
class 3 accuracy:0.8870967741935484
class 4 accuracy:0.45161290322580644
class 5 accuracy:0.9516129032258065
class 6 accuracy:0.967741935483871
class 7 accuracy:1.0
```

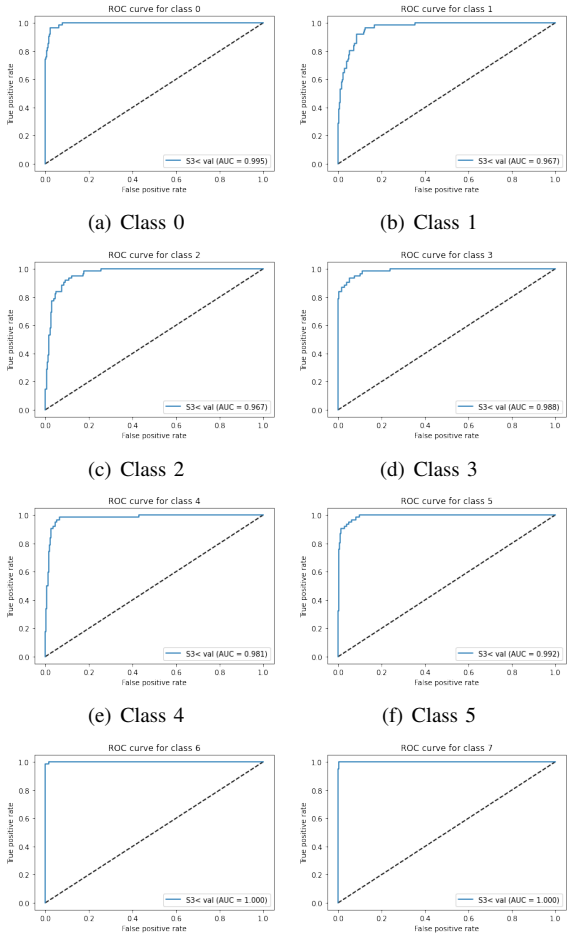
Fig. 13. Class-wise Classification Accuracy of HMT

3) AUC-ROC curve: In this part,o ur goal is to plot the AUC and ROC curves for each class. First of all, we transform the test set label to a 2D set, in which only the value in real label index is set to 1. Then we compute the FPR and TPR for each class by calling roc_curve function. After that, we fit the FPR and TPR into the funtion auc(), and obtained the returned value AUC, which is used in the plot.

Here is the code:

```
real_y=np.zeros((y_test.size,8))
for i in range(y_test.size):
   real_y[i,y_test[i]]=1
for i in range(8):
   FPR, TPR, thresholds_keras = roc_curve(
       real_y[:,i], y_pred[:,i])
   AUC = auc(FPR, TPR)
   print("AUC for class "+str(i)+": ", AUC)
   plt.figure()
   plt.plot([0, 1], [0, 1], 'k--')
   plt.plot(FPR, TPR, label='S3< val (AUC =
       {:.3f})'.format(AUC))
   plt.xlabel('False positive rate')
   plt.ylabel('True positive rate')
   plt.title('ROC curve for class '+str(i))
   plt.legend(loc='best')
   plt.show()
```

The AUC and ROC curves we plotted by executing the code are listed in in Fig. 11.



(a) Class 0 (b) Class 1

(c) Class 2 (d) Class 3

(e) Class 4 (f) Class 5

(g) Class 6 (h) Class 7

Fig. 14. Class-wise ROC Curve Plots of HMT dataset

4) normalized confusion matrix: In this part, we aim to plot the normalized confusion matrix. First, we will get the confusion matrix cm using $confusion\_matrix$ function. We use y_test which represents the real class labels of test set

and $y_predicted$ which represents the predicted labels using the model. We transformed them as $list$ to fit into the function. Then we use $cm.astype('float')/cm.sum(axis = 1)[:, np.newaxis]$ to normalize the confusion matrix. After that, we will set the plot cmap to blue confusion matrix plot, amd set the parameter in the plot ax. Finally, we will fit all the data in normalized confusion matrix into the data. Here is the code:

```
cm=confusion_matrix(list(y_test),list(
    y_predicted))
cm_normalized = cm.astype('float') / cm.sum
    (axis=1)[:, np.newaxis]
fig, ax = plt.subplots()
im = ax.imshow(cm, interpolation='nearest',
     cmap=plt.cm.Blues)
ax.figure.colorbar(im, ax=ax)
classes=[0,1,2,3,4,5,6,7]
# print(classes)
# We want to show all ticks...
ax.set(xticks=np.arange(cm_normalized.shape
    [1]),
   yticks=np.arange(cm_normalized.shape[0])
       ,
   # ... and label them with the respective
        list entries
   xticklabels=classes, yticklabels=classes
       ,
   ylabel='True label',
   xlabel='Predicted label')
plt.setp(ax.get_xticklabels(), ha="right",
         rotation_mode="anchor")
thresh = cm_normalized.max() / 2.
for i in range(cm_normalized.shape[0]):
   for j in range(cm_normalized.shape[1]):
      ax.text(j, i, format(cm_normalized[i,
           j], '.2f'),
            ha="center", va="center",
            color="white" if cm_normalized[
                i, j] > thresh else "black"
                )
fig.tight_layout()
```

The result plot is shown in Fig.15.

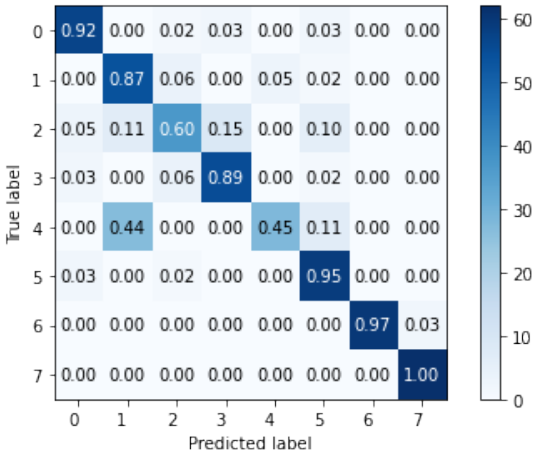


Fig. 15. Normalized Confusion Matrix Plot

5) Precision, Recall, and F-1 score: In this part, our goal is to calculate Precision, Recall, and F-1 score on the test set. We call $classification\_report()$ to compute the statistics. Here is the code:

```
print(classification_report(y_true=y_test,
    y_pred=y_predicted))
```

The result plot is shown in Fig.16.As a result, the accuracy on total test set if 0.85, the f1-score is 0.83 and the recall is 0.82.

	precision	recall	f1-score	support
0	0.89	0.92	0.90	62
1	0.61	0.87	0.72	62
2	0.79	0.60	0.68	62
3	0.83	0.89	0.86	62
4	0.90	0.45	0.60	62
5	0.78	0.95	0.86	62
6	1.00	0.97	0.98	62
7	0.97	1.00	0.98	62
accuracy			0.83	496
macro avg	0.85	0.83	0.82	496
weighted avg	0.85	0.83	0.82	496

Fig. 16. Precision, Recall, and F-1 score

B. Question 2

In this section, we describe the implementation of attribution methods application in HMT. As mentioned before, HMT is a 2-Dimensional based dataset, and therefore we need to either develop a capable XAI methods taking both 3-Dimensional and 4-dimensional tensor corresponding to 1-D and 2-D dataset, or we set up separate methods dealing with 2-Dimensional dataset, HMT. In our implementation, we choose different option for the two methods.

Our Grad-CAM++ implementation is a capable function whose code is as followed.

```
def grad_cam_plus_plus(input_model, image,
    layer_name, class_index=None):
   # if class_index is None:
   #    class_index=np.argmax(input_model.
       predict(np.array([image])), axis=-1)[0]
   """GradCAM method for visualizing input
       saliency."""
   cls = np.argmax(input_model.predict(image))
   y_c = input_model.output
   conv_output = input_model.get_layer(
       layer_name).output
   feedforward1 = keras.models.Model([
       input_model.input], [conv_output, y_c])
   # with tf.GradientTape() as tape1:
   #    with tf.GradientTape() as tape2:
   #       with tf.GradientTape() as tape3:
   #          ff_results = feedforward1([image
       ])
   #          all_fmap_masks, predictions =
       ff_results[0], ff_results[-1]
```

```
if class_index==None:
    cls=np.argmax(input_model.predict(image)
        )
else:
    cls=class_index
#          loss = predictions[:, cls]
#       grads_val = tape3.gradient(loss,
    all_fmap_masks)
#    grads_val2 = tape2.gradient(grads_val,
    all_fmap_masks)
# grads_val3 = tape1.gradient(grads_val2,
    all_fmap_masks)
with tf.GradientTape() as tape:
    ff_results=feedforward1([image])
    all_fmap_masks, predictions = ff_results
        [0], ff_results[-1]
    loss = predictions[:, cls]
grads_val = tape.gradient(loss,
    all_fmap_masks)
grads_val2=grads_val**2
grads_val3=grads_val2*grads_val
if len(image.shape) == 3:
    axis = (0, 1)
elif len(image.shape) == 4:
    axis = (0, 1, 2)
alpha_div=(2.0 * grads_val2 + grads_val3 *
    np.sum(all_fmap_masks, axis))
alpha_div = np.where(alpha_div != 0.0,
    alpha_div, 0)
alpha = grads_val2 / alpha_div
weights = np.maximum(grads_val, 0.0) *
    alpha
weights = np.sum(weights, axis=axis)
# weights = np.mean(grads_val, axis=axis)
cam = np.dot(all_fmap_masks[0], weights)
# print (cam)
H, W = image.shape[1:3]
cam = np.maximum(cam, 0)
# cam = resize(cam, (H, W))
cam = zoom(cam, H / cam.shape[0])
# cam = np.maximum(cam, 0)
cam = cam / cam.max()
return cam
```

The capability is achieved by the code of branch,

```
if len(image.shape) == 3:
    axis = (0, 1)
elif len(image.shape) == 4:
    axis = (0, 1, 2)
```

and hence it is able to process both 3-Dimensional and 4-Dimensional tensors.

Another modification needs to be mentioned is that there is a slightly modification practically in the implementation. It is modified due to troubleshooting of HMT explanation. The origin function gives an extremely unsatisfying result thus we change the way we calculate secondary gradient and third gradient. We use multiplication instead of slope to get the gradient value due to its low magnitude, the absolute value of both results are similar, however the multiplication is positive constantly.

There are some potential reasons we could come up with. First,the Grad-CAM++ are suitable for positive gradients however in both datasets, negative values take most places and therefore it it limited to perform well. Secondly, it is mentioned in the paper that Grad-CAM++ has some guided or modified variants but Our implementation works for the basic explanation. So it will be highly influence by the noise and unpredictable factors. Thirdly, among all the CAM methods, a number of the approaches choose Grad-CAM as a benchmark, so the advance Grad-CAM may also suffer from some problems which cannot adapt in multiple networks.

Subjectively, it could be clearly seen in Fig that ablation-CAM has higher contrast feature heat.

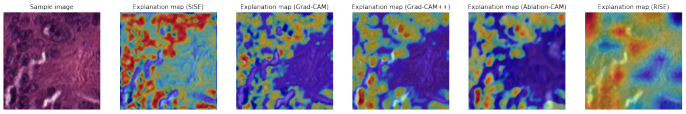


Fig. 17. Explaination Map Visualization of HMT

We implements Ablation-CAM in two separated functions, $ablation\_cam\_1d$ and $ablation\_cam\_2d$, and as the 1-D version is already described in the previous section, only the $ablation\_cam\_2d$ function is shown below.

```
def ablation_cam_2d(input_model, image,
    layer_name):

    y_c = input_model.output
    conv_output = input_model.get_layer(
        layer_name).output
    ac_model = keras.models.Model([input_model.
        input], [conv_output, y_c])

    ff_results=ac_model([image])
    all_fmap_masks, predictions = ff_results
        [0], ff_results[-1]
    # print(image.shape)
    # print(predictions)
    #index
    _pred = np.argmax(predictions, axis=1)[0]
    # print(_pred)
    # Real Y_C

    _y_c = predictions[0][_pred]
    # print(_y_c)

    # print(input_model.get_layer(layer_name))
    _wt = np.zeros(input_model.get_layer(
        layer_name).get_weights()[1].shape)
    # print(_wt)
    # print(_wt.shape)
    all_weigths = input_model.get_layer(
        layer_name).get_weights().copy()
    zero_weigth = all_weigths[0][:, :, :, 0] *
        0
    weigth_local = [np.zeros(all_weigths[0].
        shape)]
    weigth_local.append(np.zeros(all_weigths
        [1].shape))
    for i in range(_wt.shape[0]):
        weigth_local[0] = all_weigths[0].copy()
        weigth_local[0][:, :, :, i] =
            zero_weigth
```

```
        input_model.get_layer(layer_name).
            set_weights(weigth_local)
        y_k = input_model.predict([image])[0][
            _pred]
        _wt[i] = (_y_c - y_k) / _y_c

    a_k = all_fmap_masks[0]
    # print(a_k)

    ab_map = a_k * _wt
    # print(ab_map)
    explanation = np.sum(ab_map, axis=2)
    explanation = np.maximum(explanation, 0)
    # print(explanation)
    return explanation
```

The variables and parameters are defined in a naive way and influenced by the provided methods in $xai\_utils.py$. There is few difference between the functions expect for the weights calculation. The $all\_weigths$ takes different dimensional tensors so correspondingly the reference to $zero\_weigth$ are modified.

IV. Task 4: Quantitative evaluation of the attribution methods

In this task, we use two rates, decrease rate and increase rate, to evaluate the explanation map generated by the attribution methods, in order to assess the performance of the methods we implemented. Besides, we would like to use some subjective observation to evaluate the performance as a supportive evaluation.

A. Experiment

The calculator function is already provided in the notebook, which takes in a 3-Dimensional map for MNIST-1D dataset and deals with a 4-Dimensional explanation map for HMT dataset. However, we have the processing codes slightly modified, which is shown below.

An example of Grad-CAM++ rate calculation procedure in MNIST-1D dataset is as followed. Note that the explanation map should be reshape to $(40, 1)$ in order to fix in the calculator, or otherwise the explanation map has a $(40,)$ shape. We run 100 times to get a sampling result of the rates.

```
drop_rate = 0.
increase_rate = 0.
temp=np.expand_dims(x_test[0], axis=0)
print(np.expand_dims(x_test[0], axis=0).shape)
for index in range(100):
#    print(np.expand_dims(np.expand_dims(x_test
    [index], axis=0), axis=-1).shape)
    # print(index)
    prediction=model(np.expand_dims(x_test[
        index], axis=0)).numpy()
    explanation_map = grad_cam(model, np.
        expand_dims(x_test[index], axis=0),
        layer_name='conv1d_2')
    # print(explanation_map.shape)
    explanation_map = np.reshape(
        explanation_map, (40,1))
    res = calculate_drop_increase(np.
        expand_dims(x_test[index], axis=0),
        model, explanation_map, class_index=np.
        argmax(prediction[0]), frac=0.3)
    drop_rate += res[0]
    increase_rate += res[1]
drop_rate /= 100
increase_rate /= 100
print(drop_rate)
print(increase_rate)
```

An example of Grad-CAM rate calculation procedure in HMT dataset is as followed. Similar to the procedure structure in MNIST-1D, we firstly initialize the variables and reset the generator. Then recursively, we generate explanation map for 15 batches, and 32 samples per batch, and calculate both the drop and increase rate by the calculator function. To get the averaged value, which is the rate, we firstly add them up to the variable and finally divided by the total running times. Specifically, we set the parameter $frac$ to 0.9 as required in HMT datasets.

```
test_generator.reset()
drop_rate = 0.
increase_rate = 0.
for i in range(15):
    image_batch,label_batch=test_generator.next
        ()
    # print("Current Round:", i)
    for index in range(32):
        # print(i, index)
        prediction=model(image_batch)
        explanation_map_GradCAM = grad_cam(model
            , np.expand_dims(image_batch[index],
             axis=0), 'max_pooling2d_1')
        res = calculate_drop_increase(np.
            expand_dims(image_batch[index], axis
            =0), model, explanation_map_GradCAM,
             class_index=np.argmax(prediction[
            index]), frac=0.9)

        drop_rate += res[0]
        increase_rate += res[1]

drop_rate /= (15*32)
increase_rate /= (15*32)
print("======Grad-CAM======")
print(drop_rate, increase_rate)
```

Specially, for the ablation-CAM, we could only implement it and generate explanation map in the last convolution layer which is different with other maps and have a different size (112,112) correspondingly. So alternatively, the calculation process of ablation-CAM is shown as followed, in which we add a resize function to force it fit in the layer size.

```
drop_rate = 0.
increase_rate = 0.

for i in range(15):
    image_batch,label_batch=test_generator.next
        ()
    print("Current Round:", i)
```

```
    for index in range(32):
        # print(i, index)
        prediction=model(image_batch)
        explanation_map_ablation_cam =
            ablation_cam_2d(model, np.
            expand_dims(image_batch[index], axis
            =0), 'conv2d_3')
        explanation_map_ablation_cam = cv2.
            resize(explanation_map_ablation_cam,
             (224,224), interpolation=cv.
            INTER_AREA)
        res = calculate_drop_increase(np.
            expand_dims(image_batch[index], axis
            =0), model,
            explanation_map_ablation_cam,
            class_index=np.argmax(prediction[
            index]), frac=0.9)

        drop_rate += res[0]
        increase_rate += res[1]

drop_rate /= (15*32)
increase_rate /= (15*32)
print("======Ablation-CAM======")
print(drop_rate, increase_rate)
```

We can get all the rates after rounds of experiment, and the data is shown in the table.

TABLE I
DROP AND INCREASE RATES OF DIFFERENT ATTRIBUTION METHODS IN MNIST-1D AND HMT

Rate	Drop Rate (%)	Increase Rate (%)
Grad-CAM (MNIST-1D)	36.980	33.000
Grad-CAM++ (MNIST-1D)	21.964	27.000
Ablation-CAM (MNIST-1D)	19.715	32.000
Grad-CAM (HMT)	47.436	22.083
Grad-CAM++ (HMT)	50.921	19.792
Ablation-CAM (HMT)	43.341	30.469

The rates are actually metric called model truth-based metrics which measure the relationship between the explanation maps generated by attribution methods and the target model's outputs. So obviously, it could measure the performance of the attribution methods taking the explanation maps as input.

Drop rate refer to that if we remove unimportant features from the input, then the model's confidence score have no significant drop, and increase rate indicates that if we remove misleading features from the input, the model's confidence score may increase. Based on the thesis, drop rate indicate the explanation map's extent to remove unimportant features while increase rate infers the extent to remove misleading features from the input. Thus, a low drop rate and high increase rate could demonstrate a good attribution model.

From the table we could know that in HMT, Ablation-CAM works best, as it has lowest drop rate among the algorithms and highest increase rate. Besides in MNIST-1D dataset, ablation-CAM also shows greater performance than the other CAM-based attribution methods, which probably means it will be a good choice in some specific scenarios.

We will further discuss the evaluation result in the next section.

B. Discussion

This approach choose three different kind of CAM methods to evaluate two different model with different dimensions. It is obvious that Ablation-CAM holds the best performance on both datatsets. According to the paper of the ablation method, this algorithm can fit into all kinds of CNN architecture, but could perfectly fit in those networks with not so many layers. Our experiment includes one-hot and VGG-7 networks, and neither of them could be classified as DNN.

Ablation method comes out to fix the problem of gradient-related principle in the previous network. In the origin paper, it uses Grad-CAM as benchmark and use VGG and Inception-v4 as networks, resulting in a high performance in VGG but no significant improvement in Inception-v4, which is similar to our conclusion from the result.

In our evaluation, Grad-CAM++ failed to have improved performance comparing to the Grad-CAM in HMT, but have a considerable improvement on MNIST-1D. From that we could have a hypothesis that, Grad-CAM itself, including Grad-CAM++, fit into dealing with small-scale or low-dimensional problems. It can be predicted to perform well in such one-hot networks.

As a conclusion, when we need to choose an attribution method, it will be a good choice to use CAM on simple structured networks. Ablation-CAM works well in vision-related or 2-Dimensional problems, showing essential improvements comparing to the gradient-based methods. However, there is also an important drawback of Ablation-CAM that, it has much longer running time than the others due to its complete visit to every unit in every maps. In those small-scaled problems, especially one-dimensional based problem, Grad-CAM++ will take great advantages of it.

REFERENCES

[1] Selvaraju, Ramprasaath R., Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. "Grad-cam: Visual explanations from deep networks via gradient-based localization." In Proceedings of the IEEE international conference on computer vision, pp. 618-626. 2017. https://arxiv.org/abs/1610.02391

[2] Chattopadhay, Aditya, Anirban Sarkar, Prantik Howlader, and Vineeth N. Balasubramanian. "Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks." In 2018 IEEE winter conference on applications of computer vision (WACV), pp. 839-847. IEEE, 2018. https://arxiv.org/abs/1710.11063

[3] Ramaswamy, Harish Guruprasad. "Ablation-cam: Visual explanations for deep convolutional network via gradient-free localization." In Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision, pp. 983-991. 2020.

[4] Franelstar. "Franelstar/Heatmap-CNN". *github*.(2021) [Online]. Available:https://github.com/Franelstar/Heatmap-CNN