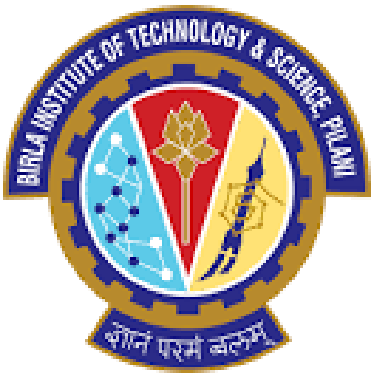

Designing Accuracy Metrics for Lung Cancer Detection using CT Scan Images

Harsh Swapnil Deshpande (2021A7PS2225P)
Suri Sai Viswanadha Aditya (2020A7PS0077P)
Arvin Datta (2021A7PS0430P)

April 2023

Developing accuracy metrics to evaluate the performance of a CNN model for detecting lung cancer using CT scan images.

Harsh Deshpande (2021A7PS2225P)
Suri Sai Viswanadha Aditya (2020A7PS0077P)
Arvin Datta (2021A7PS0430P)

April 2023

Acknowledgements

We would like to express our sincere gratitude to Dr. Vishal Gupta for his invaluable guidance and support throughout the project. The knowledge and experience that we gained under his mentorship were instrumental in shaping our understanding of the subject and enabled us to develop this disease detection model.

It has allowed us to reflect upon the conventional methods of object detection and develop new metrics of determining accuracy. This project has helped us in gaining a deeper insight of the subject, and has given us an opportunity to build a real world model and gain substantial experience.

Finally, we would like to thank our families and friends for their unwavering support and encouragement throughout this project.

1 Overview and Implementation details

Lung cancer is one of the most common and deadly forms of cancer worldwide, and early detection is crucial for effective treatment and improved patient outcomes. In recent years, deep learning algorithms have shown promise in detecting lung cancer using computed tomography (CT) scan images. However, accurately assessing the performance of these algorithms is critical to ensure their reliability in clinical settings.

This is where accuracy metrics come into play. Accuracy metrics are essential tools for evaluating the performance of deep learning models in detecting lung cancer using CT scan images. These metrics enable researchers and healthcare professionals to quantitatively assess the accuracy, sensitivity, and specificity of these algorithms, allowing them to identify strengths and weaknesses and make necessary improvements.

By using accurate metrics to evaluate deep learning algorithms, we can improve their reliability and ensure that they are effective tools for detecting lung cancer at an early stage. Ultimately, this can lead to better patient outcomes and a reduction in lung cancer-related deaths

In this project we have studied the existing accuracy metrics analysed their performance and then suggested certain novel accuracy metrics. We have also tested the performance of all these metrics on the LIDC-IDRI dataset, which we annotated on the basis of severity on a scale of 0 to 9. However, our ideas for the accuracy metrics are independent of any model or dataset, although performance may for obvious reasons vary. We have come up with 3 main metrics 1)Weighted accuracy 2)Biased accuracy 3)WCPDI(Weighted Class Probability Distribution) 4)DWA (Directional Weighted Accuracy) We have primarily used the torchvision,numpy, and sklearn libraries and packages from the same. For plotting the graphs we have used matplotlib.

The code to our colab notebook can be found here for however is interested. You can add whichever dataloader you wish so as to run the metrics for your custom dataset as well.

Link to colab notebook Notebook **It must be noted that to run any of our metrics on your required dataset, you must write your own dataloader for the same, in the code we have provided you must insert any dataloader for a dataset which has annotated images from 0 to 9.**

2 Literature Review

The article "Multi-Disease Prediction Based on Deep Learning: A Survey" provides a comprehensive understanding of deep learning (DL) applications in healthcare. The authors start by defining DL and discussing its advantages and limitations. They then review DL applications in various healthcare areas, including disease diagnosis, medical imaging, electronic health records (EHRs), and drug discovery.

The article highlights the importance of DL in improving disease diagnosis accuracy, particularly in the case of cancer detection, where it has been shown to outperform human experts. In medical imaging, DL has been applied to identify and segment organs and tissues, and to detect abnormalities in various modalities, including X-rays, CT scans, and MRI scans.

The paper discusses several models based on neural networks, and their subsequent functionalities in the detection of various kinds of diseases. The working and architectures of some artificial neural networks such as CNN, RNN, LSTM and GAN have also been briefly explained.

The authors also review DL applications in EHRs, where DL models have been used to predict patient outcomes and identify potential adverse events, such as hospital readmissions. DL has also been applied in drug discovery to predict the effectiveness and toxicity of new drugs.

Overall, the article concludes that DL has enormous potential to transform healthcare by improving disease diagnosis, enhancing medical imaging analysis, and supporting clinical decision-making. However, the authors also note that there are several challenges to overcome, including the need for large, high-quality datasets, and the potential for biased models.

In summary, this literature review provides an informative overview of the current state of DL applications in healthcare, highlighting both the opportunities and challenges in this field. It would be of interest to researchers and healthcare professionals interested in DL and its potential applications in healthcare.

3 Background

The concepts that have been involved in the culmination of this project and can be considered to be prerequisites for the project include:

1. **Convolutional Neural Networks (CNN)**: CNN is a class of artificial neural network which is used for the analysis of visual objects or images. It is based on the convolution of two functions, a mathematical operation which determines the modification of the graph of one function due to another. CNN consists of an input layer, hidden layers, and an output layer. The convolutions are performed by these hidden layers which in turn contribute to the next layer.

2. **Confusion Matrix**: A confusion matrix is a matrix with two dimensions, actual and predicted. The terminologies involved are:

    - Condition positive (P) - the number of actual positive cases in the dataset
    - Condition negative (N) - the number of actual negative cases in the dataset
    - True positive (TP) - correct prediction of an attribute that is actually present

- True negative (TN) - correct prediction of an attribute that is actually absent
- False positive (FP) - incorrect prediction of an attribute that it is present when it is actually absent
- False negative (FN) - incorrect prediction of an attribute that it is absent when it is actually present

Some important metrics derived from the confusion matrix are:

- Precision = TP / (TP + FP)
- Accuracy = (TP + TN) / (TP + TN + FP + FN)

3. **Gaussian Kernel Distribution Estimation (KDE)**: Kernel density estimation is a way to estimate the probability density function (PDF) of a random variable in a non-parametric way. KDE using Gaussian kernels works for both univariate data like a simple array or multivariate data like an array of more than one dimension. It estimates the probability density function by using a kernel (a mathematical function) to smooth the data. The kernel is centered at each data point and summed to create a curve, which is typically a Gaussian distribution. It assigns more weight to data points that are closer to the center. For example, if the curve is higher at a certain value, that means that value occurs more frequently in the data.

.

3.1 Existing Metrics

In this code snippet, several evaluation metrics are computed to assess the performance of a classification model. These metrics and their explanations are as follows:

1. **Accuracy**: This metric measures the proportion of correct predictions out of the total number of predictions made.
2. **Cohen's Kappa**: This metric measures the agreement between two raters beyond chance. It takes into account the possibility of random agreement.
3. **Matthews Correlation Coefficient (MCC)**: This metric is used to measure the quality of binary classifications. It takes into account true positives, false positives, true negatives, and false negatives and returns a value between -1 and +1.
4. **Precision**: This metric measures the proportion of true positives among the samples that were predicted as positive.
5. **Recall**: This metric measures the proportion of true positives among all positive samples.

6. **Log loss**: This metric is used to evaluate the performance of a classification model that outputs probabilities of different classes.

7. **F1 score**: This metric is the harmonic mean of precision and recall, and it provides a balance between the two metrics.

8. **Weighted F1 score**: This metric is similar to the F1 score but takes into account the class imbalance by weighting each class by its proportion in the dataset.

9. **Confusion matrix**: This matrix shows the number of true positive, false positive, true negative, and false negative predictions for each class.

10. **Precision-recall curve**: This curve shows the trade-off between precision and recall for different probability thresholds.

11. **Multiclass ROC-AUC curve**: This curve shows the trade-off between true positive rate and false positive rate for different probability thresholds in a multiclass classification problem.

These metrics are important for evaluating the performance of a classification model and understanding which classes are misclassified. They can also be used to select an appropriate threshold for a classification model based on the desired precision and recall values.

Below we show a snippet of the code which we have implemented to compute these existing metrics.

```
acc = accuracy_score(y_true, y_pred)
print('Accuracy: {:.4f}'.format(acc))

kappa = cohen_kappa_score(y_true, y_pred)
print("Cohen's Kappa:", kappa)

mcc = matthews_corrcoef(y_true, y_pred)
print("Matthew's Correlation Coefficient (MCC): {:.3f}".format(mcc))

precision = precision_score(y_true_one_hot, y_pred_one_hot, average='macro')
print('Precision: {:.4f}'.format(precision))

recall = recall_score(y_true_one_hot, y_pred_one_hot, average='macro')
print('Recall: {:.4f}'.format(recall))

logloss = log_loss(y_true_one_hot, y_pred_one_hot)
print("Log loss:", logloss)

f1 = f1_score(y_true_one_hot, y_pred_one_hot, average='macro')
print('F1 score: {:.4f}'.format(f1))

weighted_f1 = f1_score(y_true_one_hot, y_pred_one_hot, average='weighted')
print('Weighted F1 score: {:.4f}'.format(weighted_f1))

# Compute confusion matrix and plot it
cm = confusion_matrix(y_true, y_pred)
```

```
plt.figure()
plot_confusion_matrix(cm, classes=[str(i) for i in range(10)], normalize=True, title='Confusion matrix')
plt.show()

# Compute precision-recall curve and plot it
precision, recall, _ = precision_recall_curve(y_true_one_hot.ravel(), y_pred_one_hot.ravel())
plt.figure()
plt.plot(recall, precision, color='b', alpha=0.2)
plt.fill_between(recall, precision, step='post', alpha=0.2, color='b')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall curve')
plt.show()
y_one_hot = label_binarize(y_true, classes=np.arange(10))

plot_multiclass_roc_auc(y_true, y_pred, n_classes=10)
```

```
Train Epoch: 1 [0/60000 (0%)]   Loss: 2.304257
Train Epoch: 1 [6400/60000 (11%)]       Loss: 1.613831
Train Epoch: 1 [12800/60000 (21%)]      Loss: 1.566616
Train Epoch: 1 [19200/60000 (32%)]      Loss: 1.509639
Train Epoch: 1 [25600/60000 (43%)]      Loss: 1.522662
Train Epoch: 1 [32000/60000 (53%)]      Loss: 1.573924
Train Epoch: 1 [38400/60000 (64%)]      Loss: 1.564526
Train Epoch: 1 [44800/60000 (75%)]      Loss: 1.535100
Train Epoch: 1 [51200/60000 (85%)]      Loss: 1.571709
Train Epoch: 1 [57600/60000 (96%)]      Loss: 1.517466
Average loss: 1.5626
Average loss: 1.5168
Accuracy: 0.9460
Cohen's Kappa: 0.9399737625316026
Matthew's Correlation Coefficient (MCC): 0.940
Precision: 0.9466
Recall: 0.9451
Log loss: 1.9463572830123281
F1 score: 0.9454
Weighted F1 score: 0.9460
Normalized confusion matrix
```

Figure 1: Displaying the Metrics as calculated on colab

The above image show the values of all the standard accuracy metrics as calculated on the training dataset. We have also shown the loss as calculated per epoch while training.

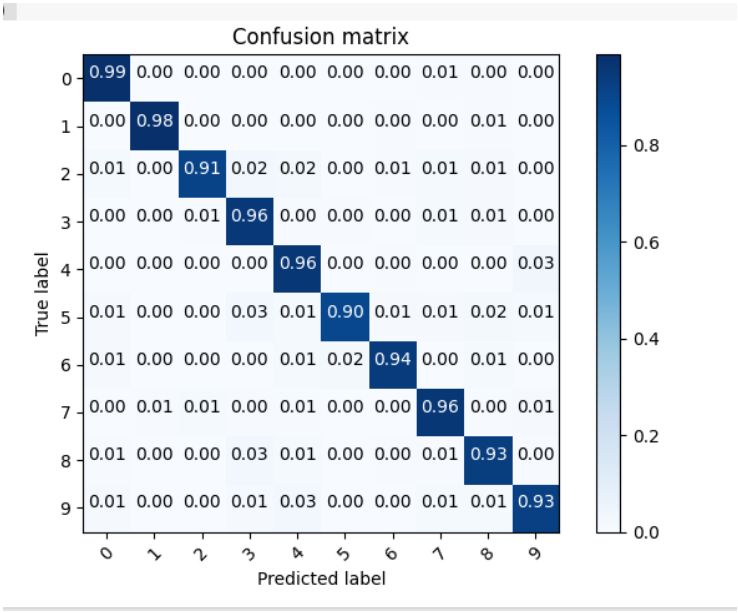

Figure 2: Confusion Matrix

The above image displays the Confusion Matrix for our model and dataset. The model gives us an idea of how many predicitions have been made above the actual value, how many have been made below and how many have been made correctly.On the x-axis we have the predicted label for severity of the lung cancer detected (0 to 9) and on the y-axis we have the true label for the same. The matrix represents the percentage of predictions for each label in the entries of the matrix.

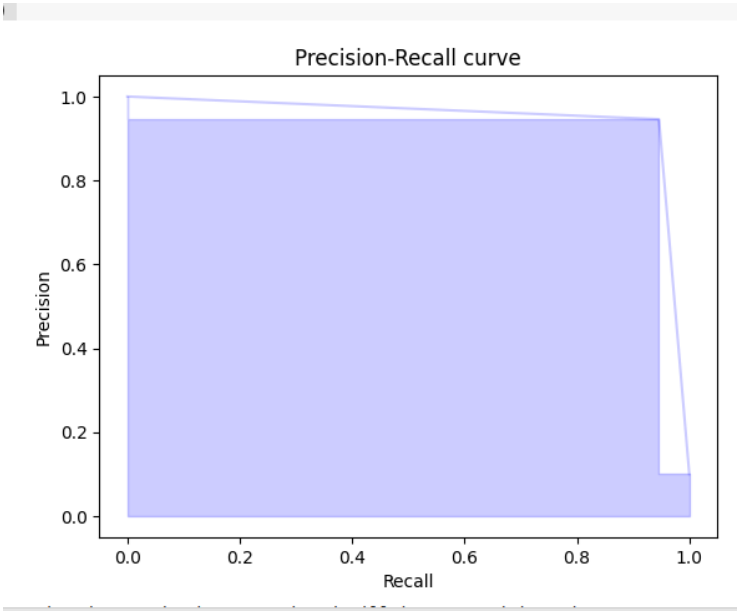

Figure 3: Precision-Recall curve

The above plot displays the precision-recall curve for our dataset and model. The precision-recall curve plots precision on the y-axis and recall on the x-axis, using a range of thresholds for the model. Each point on the curve represents a different threshold, and the curve is created by connecting these points.

The intuition behind the PR curve is that a classifier with perfect precision and recall would achieve a point at (1,1) on the curve, indicating that it correctly classified all positive examples and made no false positive predictions. However, in practice, there is often a trade-off between precision and recall: increasing the threshold to improve precision may result in a decrease in recall, and vice versa.

In an imbalanced dataset, where one class has significantly fewer examples than the other, a classifier may achieve high accuracy by simply predicting the majority class for all examples. However, this approach may result in poor recall and precision for the minority class. The PR curve can help to identify the optimal threshold for the classifier, which balances the precision and recall for both classes. A classifier with a PR curve that is closer to the top-right corner of the plot is considered to be better than one with a curve that is closer to the bottom-left corner. The above plot if for the multi-class ROC curve for our dataset and model. OC (Receiver Operating Characteristic) curve is a graphical representation that shows the diagnostic performance of a binary classifier system at different classification thresholds. It is a plot of the True Positive Rate (TPR) against the False Positive Rate (FPR) at different threshold settings. The area under the ROC curve (AUC-ROC) is a widely used metric

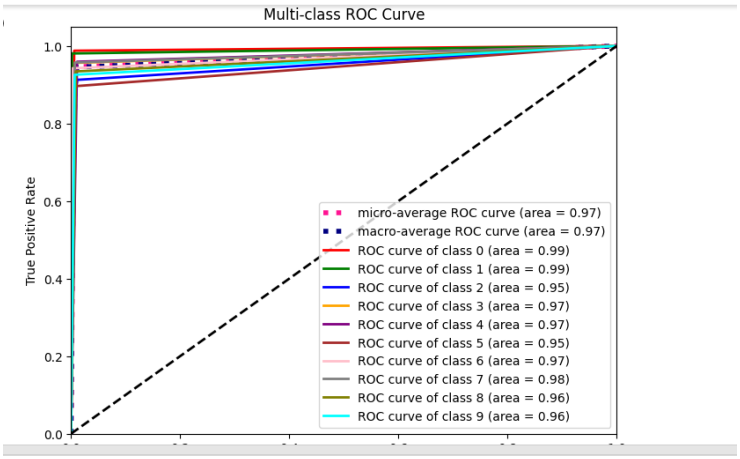

Figure 4: Multi-Class ROC

for evaluating the performance of binary classification algorithms.

In a multiclass classification problem, where there are more than two classes, the ROC curve can be extended to a MultiClass ROC curve. The MultiClass ROC curve is a generalization of the binary ROC curve to evaluate the performance of a multi-class classification system. In the MultiClass ROC curve, the TPR and FPR are computed for each class by comparing the positive instances of that class against the negative instances of all the other classes. The area under the MultiClass ROC curve (AUC-MROC) is used to evaluate the overall performance of the multi-class classification system.

The intuition behind the MultiClass ROC curve is similar to the binary ROC curve. It helps to visualize the trade-off between the true positive rate and the false positive rate for different classification thresholds. The MultiClass ROC curve provides a comprehensive evaluation of the classification system performance across all classes simultaneously. The higher the AUC-MROC, the better the overall performance of the multi-class classification system.

In the context of lung cancer detection using CT scan images, a MultiClass ROC curve can be used to evaluate the performance of a multi-class classification system that rates the severity of the cancer on a scale of 0 to 9. The MultiClass ROC curve can help to determine the sensitivity and specificity of the system at different thresholds of the severity rating, which can aid in identifying the optimal classification threshold to maximize the system's performance. The AUC-MROC can be used to compare the performance of different classification models and to select the best performing model for the lung cancer severity rating task.

4 Weighted Accuracy

This is the simplest metric we have implemented which simply attempts to naively give higher priority to higher severity of the disease.

$$\text{weighted\_accuracy} = \frac{\sum_{i=1}^{10} w_i p_i}{\sum_{i=1}^{10} w_i}$$

The weighted accuracy metric is a performance measure that takes into account the distribution of classes in a dataset by assigning different weights to each class based on its importance. In some datasets, certain classes may be more important than others, or the cost of misclassifying one class may be higher than the cost of misclassifying another. Weighted accuracy is a way to account for these differences by giving more weight to the classes that are more important or have a higher cost of misclassification.

To calculate weighted accuracy, first, we calculate the accuracy for each class separately by dividing the number of correct predictions for that class by the total number of samples for that class. Then, we multiply each class accuracy by its corresponding weight and take the average to get the weighted accuracy score. The weights can be predefined based on domain knowledge or learned from the data.

For example, in our lung cancer dataset with ten classes representing different severity, and misclassifying a rare and severe tumour/cancer has a higher cost than misclassifying a common and mild one. In that case, we can assign higher weights to the severe classes and lower weights to the common and mild classes. The weighted accuracy metric will then reflect the importance of correctly predicting the rare and severe classes and penalize misclassifying them more severely than the common and mild diseases. The code for the same is as follows

```python
import torch

def weighted_accuracy(predictions, targets):
# Define the weights for each severity
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
weights = torch.tensor([1/45, 2/45, 3/45, 4/45, 5/45, 6/45, 7/45, 8/45, 9/45, 10/45], device=device)
# Move input tensors to the same device as the weights tensor
predictions = predictions.to(device)
targets = targets.to(device)

# Calculate the proportion of correctly predicted images for each severity
num_correct = torch.zeros(10, device=device)
total = torch.zeros(10, device=device)
for i in range(10):
    mask = (targets == i)
    num_correct[i] = torch.sum(predictions[mask] == i)
    total[i] = torch.sum(mask)
proportions = num_correct / total

# Calculate the weighted accuracy score
weighted_proportions = weights * proportions
```

```
    weighted_accuracy = torch.sum(weighted_proportions) / torch.sum(weights)

    return weighted_accuracy.item()
```

5 Biased Accuracy

We have implemented a novel accuracy metric, called "biased accuracy", which is designed to take into account the severity of disease predicted by a machine learning model when evaluating its performance. The metric is particularly useful in the context of lung cancer detection using CT scan images, where the severity of disease predicted by the model can have important implications for patient care.

$$\text{weight} = \begin{cases} \alpha \cdot \dfrac{(t-p)^2}{t-1} & \text{if } p < t \\ 1 & \text{if } p = t \\ \dfrac{1}{1+|p-t-d|} & \text{if } p > t \end{cases}$$

$$\text{biased\_accuracy} = \frac{\sum_{i=1}^{n} w_i \cdot [y_i = \hat{y}_i]}{\sum_{i=1}^{n} w_i}$$

where t is the true severity level, p is the predicted severity level, d is a parameter that controls the degree of bias, α is a hyperparameter that controls the strength of the penalty for underestimating severity, w_i is the weight assigned to the ith prediction, n is the total number of predictions, y_i is the true label of the ith prediction, and $\hat{y}_i$ is the predicted label of the ith prediction

The biased accuracy metric is calculated by assigning weights to correct and incorrect predictions based on the difference between the predicted and true severity of disease. If the predicted severity is lower than the true severity, the weight is proportional to the square of the difference between them, divided by the true severity minus one, and multiplied by a constant alpha. This encourages the model to predict a higher severity of disease when appropriate, as lower severity predictions are more heavily penalized. If the predicted severity is the same as the true severity, the weight is set to one. Finally, if the predicted severity is higher than the true severity, the weight is inversely proportional to the absolute difference between them, plus a constant offset d.

The biased accuracy metric is computed for each instance in the test set, and the weights are used to update a running total of correct predictions. The metric is biased because it gives more weight to correct predictions that are consistent with a higher severity of disease. To obtain a more general estimate of the model's performance, the biased accuracy samples are passed through a kernel density estimation algorithm to generate a smoothed density function.

In the context of lung cancer detection using CT scan images, the biased accuracy metric is valuable because it can help identify cases where the model

is not predicting a high enough severity of disease. For example, a patient with a high-risk lesion may require more aggressive treatment than a patient with a low-risk lesion, so it is important for the model to accurately predict the severity of disease. The biased accuracy metric can be used to evaluate the model's performance on a test set of CT scan images and provide feedback for model improvement. The code as we have implemented it can be found below.

```python
def biased_accuracy(y_true, y_pred, d, alpha=1, num_samples=1):
    acc = 0
    total_weight = 0
    samples = []
    for i in range(len(y_true)):
        true_val = y_true[i]
        pred_val = y_pred[i]

        if isinstance(pred_val, list):
            pred_val = max([val for val in pred_val if val < 11])
        if pred_val < true_val:
            weight = alpha * (true_val - pred_val) ** 2 / (true_val - 1)
        elif pred_val == true_val:
            weight = 1
        else:
            weight = 1 / (1 + abs(pred_val - true_val - d))

        acc += weight * (pred_val == true_val)
        total_weight += weight
        samples.append(acc / total_weight)

    # Apply kernel density estimation to the biased accuracy samples
    kde = gaussian_kde(samples)
    x = np.linspace(np.min(samples), np.max(samples), num_samples)
    y = kde(x)

    return y
```

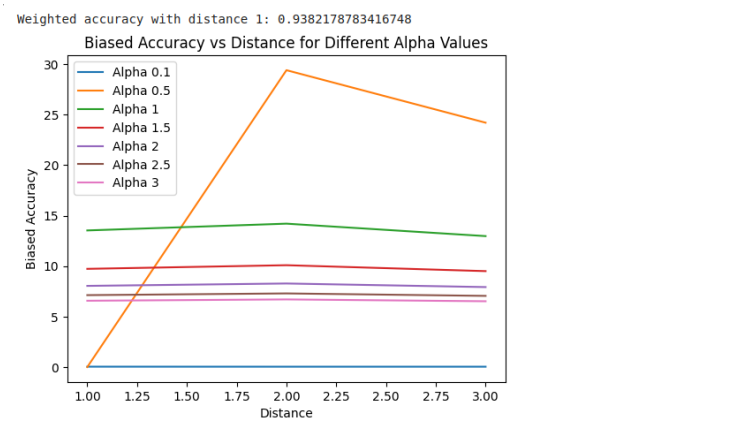

Figure 5: Biased-Accuracy vs Distance values

In order to correctly model our requirements we need a graph which gives a maxima at the correct value and has a higher downward slope towards lower values so as to model the bias for higher valued incorrect predictions. This is perfectly depicted in the plot with $\alpha = 0.5$.

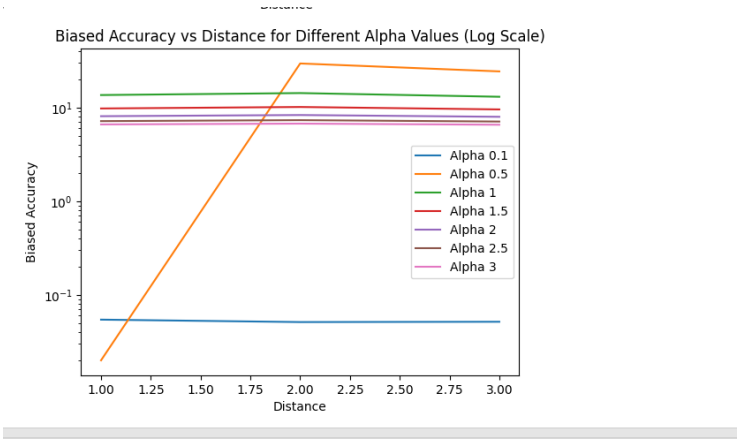

Figure 6: Biased-Accuracy vs Distance values

On analysing the log scaled plot we again arrive at the same conclusion. Again we observe that the sharpest peak and greatest downward sloping curve

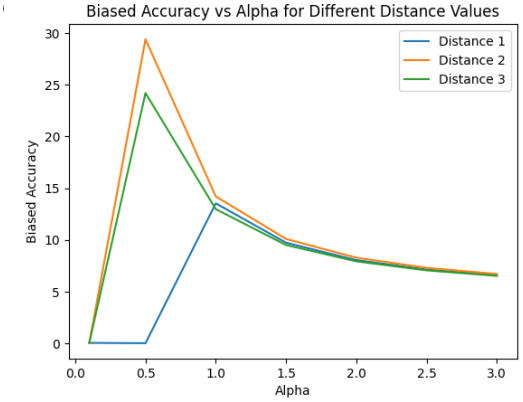

Figure 7: Biased-Accuracy vs Alpha values

is obtained for distance 2. Hence the optimal distance is 2 The same is also observed via the log scaled plot.

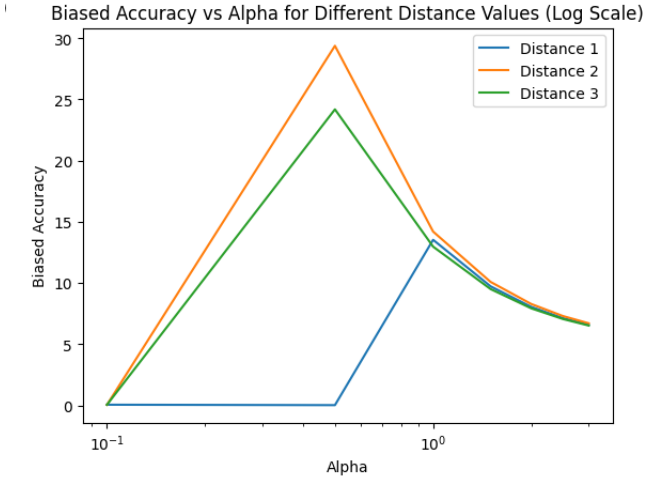

Figure 8: Biased-Accuracy vs Alpha (log scale) values

Why use Gaussian KDE?

Gaussian KDE stands for Gaussian Kernel Density Estimation. It is a non-parametric way of estimating the probability density function (PDF) of a random variable. Given a set of observations, Gaussian KDE can estimate the underlying probability density function by computing the kernel density estimate. In other words, Gaussian KDE is a way of estimating the probability density function of a continuous random variable, by smoothing the observations and assigning a probability density to each point based on the observations around it.

The procedure for estimating the PDF using Gaussian KDE involves the following steps:

1. Choose a kernel function, usually a symmetric and non-negative function such as the Gaussian kernel.

2. Choose a bandwidth parameter, which determines the amount of smoothing. A small bandwidth results in a high variance estimate while a large bandwidth results in a low variance estimate.

3. For each observation in the data, place a kernel function around it and weight it by the bandwidth parameter. The weighted kernel function is then summed over all the observations to get the estimated probability density function.

Gaussian KDE is particularly useful when the data is not easily modeled by a simple parametric distribution, such as a normal distribution. It allows for a more flexible estimation of the probability density function, which can then

be used for various statistical analyses, such as hypothesis testing or confidence intervals.

In the code snippet provided, Gaussian KDE is used to estimate the probability density function of the biased accuracy samples generated by the biased_accuracy function. The gaussian_kde function from the scipy.stats library is used to perform the kernel density estimation. The resulting estimate is then plotted as a smooth curve over a range of values using the numpy library.

6 Weighted CPDI

We have calculated the Weighted Class Probability Distribution Index (WCPDI) for a given set of true and predicted labels. The WCPDI is a measure of how well the predicted probabilities of a model match the true probabilities. It is an extension of the Class Probability Distribution Index (CPDI) that takes into account the class imbalance in the dataset.

$$\text{WCPDI} = \frac{1}{K} \sum_{i=1}^{K} w_i \sigma_i$$

The CPDI is defined as the Euclidean distance between the true class probabilities and the predicted class probabilities, normalized by the maximum possible distance between the two distributions. The CPDI ranges from 0 to 1, where a value of 0 indicates a perfect match between the true and predicted probabilities, and a value of 1 indicates no match.

The WCPDI takes into account the class imbalance by assigning weights to each class proportional to the inverse of its frequency in the dataset. The weights are used to compute the standard deviation of the predicted probabilities for each class, which is then multiplied by the weight and averaged over all classes to obtain the WCPDI.

The code defines a weighting function 'w' that takes in the predicted probabilities 'p' and the true labels 'y' as input and computes the weights for each sample. It then computes the weighted predicted probabilities for each class and the standard deviation of the weighted probabilities, which are used to calculate the WCPDI.

```

def compute_wcpdi(true_labels, pred_labels, k=1):
    # Define weighting function
    def w(p, y):
        batch_size = p.shape[0]
        p = p.unsqueeze(1).repeat(1, 10, 1)
        y_onehot = F.one_hot(y, num_classes=10).unsqueeze(-1)
        dists = pairwise_distances(p.reshape(-1, p.size(-1)), y_onehot.reshape(-1, y_onehot.size(-1)))
        dists_tensor = torch.from_numpy(dists).reshape(batch_size, -1)
        return torch.exp(-k * dists_tensor)

```

```
    # Convert predicted labels to float data type
    pred_labels = pred_labels.float()

    # Compute predicted probabilities for all test samples
    probs = F.softmax(pred_labels, dim=1)

    # Compute standard deviation of weighted predicted probabilities for each class
    weights = torch.empty(10, dtype=torch.float32)
    stddevs = torch.empty(10, dtype=torch.float32)
    for i in range(10):
        idx = (true_labels == i).nonzero(as_tuple=True)[0]
        if len(idx) > 0:
            p = probs[idx]
            y = true_labels[idx]
            weight = w(p, y)
            weights[i] = weight.mean()
            weighted_p = weight.unsqueeze(-1) * p.unsqueeze(1)
            centered_p = weighted_p - weighted_p.mean(dim=0)
            variances = (centered_p.pow(2).sum(dim=0) / weight.sum()).mean(dim=0)
            stddevs[i] = variances.sqrt().mean()
        else:
            weights[i] = 0
            stddevs[i] = 0

    # Compute weighted CPDI
    wcpdi = (stddevs * weights).mean()

    return wcpdi
```

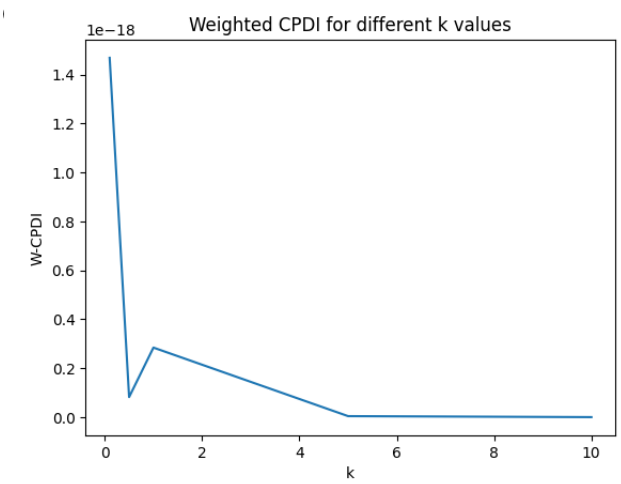

Figure 9: Weighted CPDI

For selecting the optimal k value we can use the following reasoning. We should select a k value which if shifted makes a significant difference to the WCPDI value obtained, this will ensure that our final metric values will remain

sparse and hence will be able to distinguish between even similar results. Therefore a value between 1 and 2 should be optimal based on the graph obtained. .

7 Directional Weighted Accuracy

The proposed metric, Directional Weighted Accuracy (DWA), can be helpful in evaluating deep learning models for the task of lung cancer detection with 10 labels of severity.

In this task, the goal is to predict the severity of lung cancer based on medical imaging data. The severity labels can range from 0 (no cancer) to 9 (most severe cancer), with intermediate levels of severity in between. The task is challenging due to the subtle differences in image features that can indicate different levels of severity, and the potential overlap between different severity levels.

DWA is a metric that can take into account the direction of errors in the predictions. It is defined as:

$$DWA = \frac{\sum_{i=0}^{9} TP_i \cdot w_i}{\sum_{i=0}^{9} (TP_i + FP_i) \cdot w_i} \tag{1}$$

where TP_i and FP_i are the numbers of true positives and false positives for severity level i, respectively, and w_i is a weight factor that depends on the direction of the error.

In particular, the weight factor for level i is defined as:

$$w_i = \begin{cases} \alpha, & \text{if } TP_i > FP_i \\ \beta, & \text{if } TP_i < FP_i \\ 1, & \text{if } TP_i = FP_i \end{cases} \tag{2}$$

where α and β are hyperparameters that control the bias towards false positives or false negatives, respectively. When $\alpha > \beta$, the metric is biased towards false positives, which means that it is more important to correctly predict higher severity levels than lower severity levels. Conversely, when $\beta > \alpha$, the metric is biased towards false negatives, which means that it is more important to correctly predict lower severity levels than higher severity levels.

By plotting the DWA values for different combinations of α and β, and for varying values of each hyperparameter, we can gain insights into the behavior of the deep learning model and its performance for different levels of severity. For example, if the DWA values are highest for a combination of α and β that is biased towards false positives, it may suggest that the model is better at predicting higher severity levels than lower severity levels. Conversely, if the DWA values are highest for a combination of α and β that is biased towards false negatives, it may suggest that the model is better at predicting lower severity levels than higher severity levels.

In summary, DWA is a useful metric for evaluating deep learning models for the task of lung cancer detection with 10 labels of severity. By taking into account the direction of errors and biasing the metric towards false positives or false negatives, we can gain insights into the model's behavior and optimize its performance for different levels of severity.

```python
def compute_dwa(y_true, y_pred, alpha=1.0, beta=1.0):

#Computes the Directional Weighted Accuracy (DWA)
#for a list of true labels and predicted labels.
#alpha and beta are hyperparameters that
#control the strength of the directional weight.

# Convert lists to tensors
y_true = torch.tensor(y_true)
y_pred = torch.tensor(y_pred)

# Compute the confusion matrix
conf_mat = torch.zeros((10, 10))
for i in range(len(y_true)):
    conf_mat[y_true[i], y_pred[i]] += 1

# Compute the weight matrix
weight_mat = torch.zeros((10, 10))
for i in range(10):
    for j in range(10):
        weight_mat[i, j] = (1 + alpha) / (1 + beta * abs(i-j))

# Compute the DWA
weighted_conf_mat = torch.mul(conf_mat, weight_mat)
dwa = torch.sum(weighted_conf_mat) / torch.sum(conf_mat)

return dwa.item()
```

This heat map explains the variation of the DWA metric with the α and β

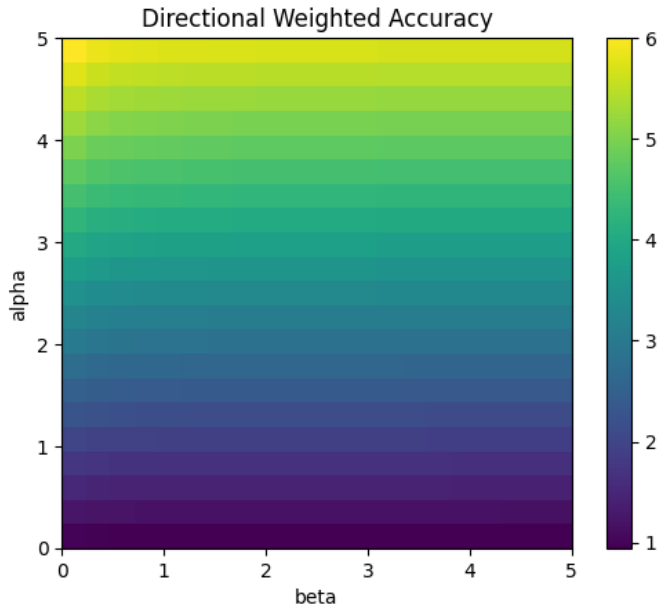

Figure 10: Heat Map

values. A yellowish colour represents a higher value and a more violet colour represents a lower value. We wish to choose an α value which has a correspondingly thin range. So alpha values of around 3 should be optimal. However, such a variation cannot be seen over beta values as they are linearly distributed as can be seen from the plots below as well. The above graphs have been plotted

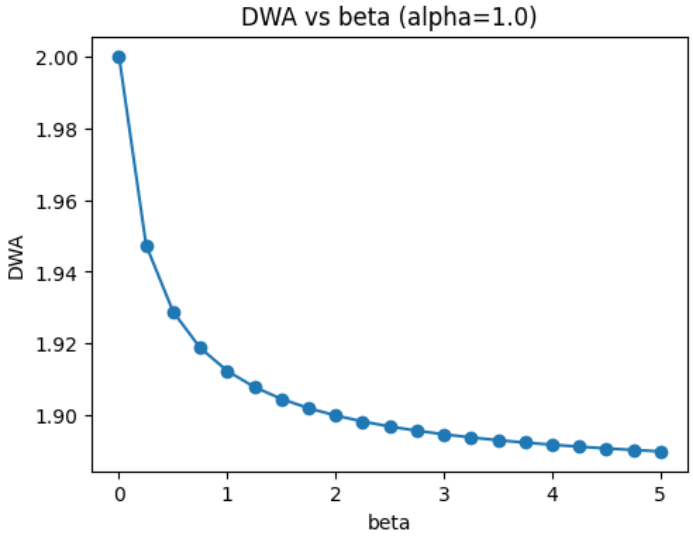

Figure 11: DWA vs Beta (alpha=1)

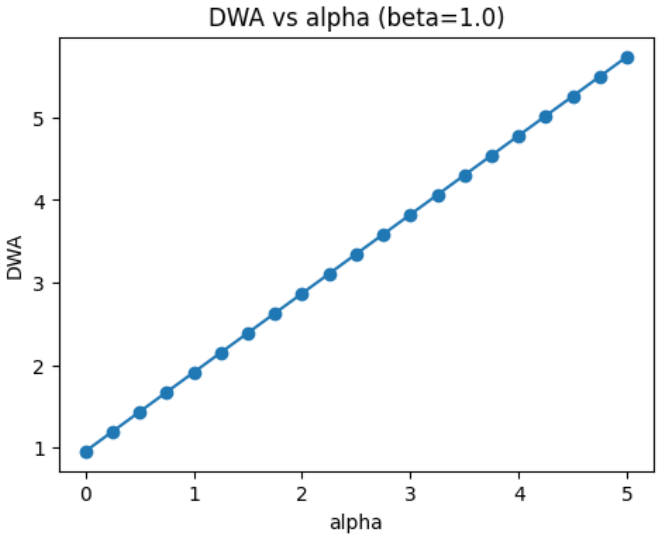

Figure 12: DWA vs Alpha (beta=1)

so as to study the variation of the nature of the curves with both alpha and beta values. However, as we can clearly see from the plots above the nature of the curve remains the same, i.e, linear curve for DWA vs alpha and monotonically

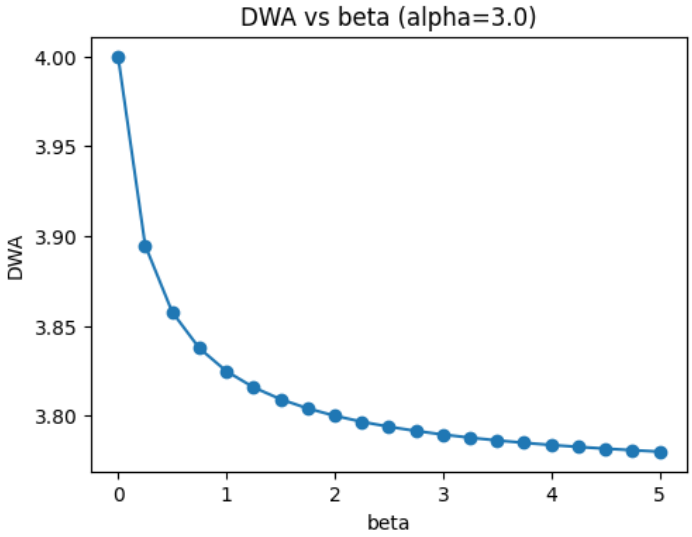

Figure 13: DWA vs Beta (alpha=3)

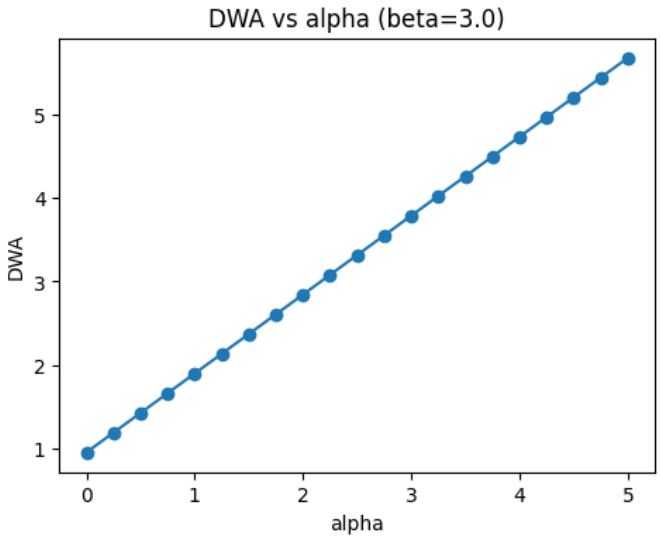

Figure 14: DWA vs Alpha (beta=3)

decreasing for DWA vs beta.

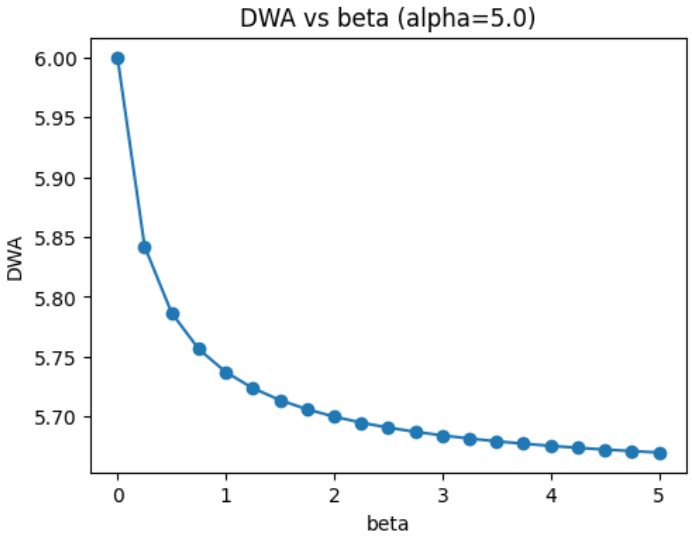

Figure 15: DWA vs Beta (alpha=5)

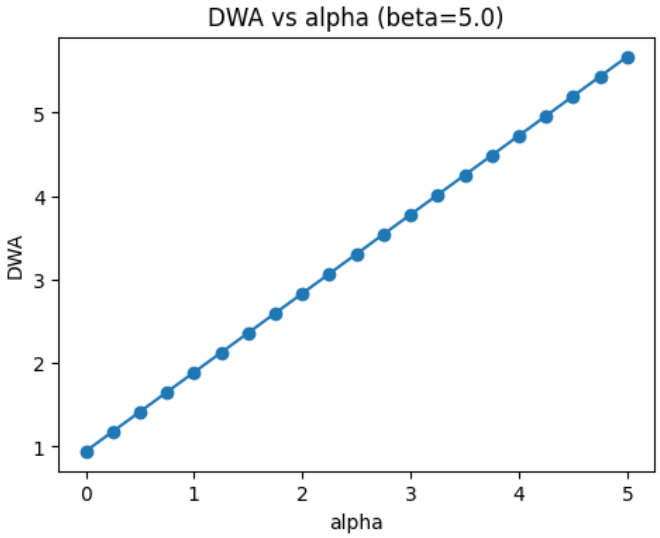

Figure 16: DWA vs Alpha (beta=5)

References

Valueva, M. V., Nagornov, N., Lyakhov, P. A., Valuev, G., & Chervyakov, N. I. (2020). Application of the residue number system to reduce hardware costs of

the convolutional neural network implementation. Mathematics and Computers in Simulation, 177, 232–243. doi: 10.1016/j.matcom.2020.04.031

Xie, S., Yu, Z., & Lv, Z. (2021). Multi-Disease Prediction Based on Deep Learning: A Survey. CMES-Computer Modeling in Engineering & Sciences, 128(2), 489–522. doi: 10.32604/cmes.2021.01672