Creating Custom Loss Functions for Multiclass Classification

Researcher: Yousuf Rehman, Advisors: Patrick Gray, Guillermo Sapiro

Duke Bass Connections: Deep Learning & Remote Sensing for Coastal Resilience

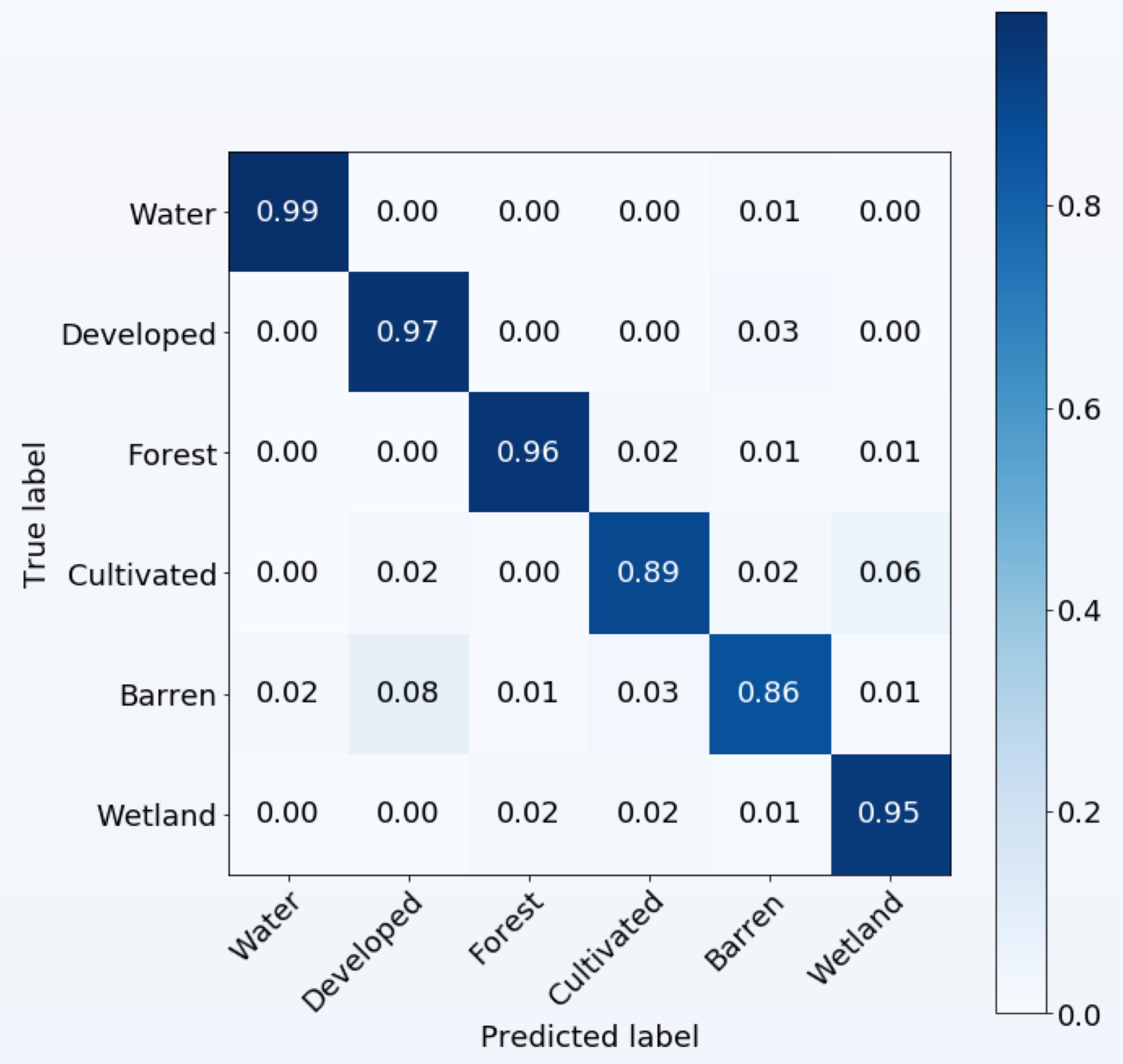
INTRODUCTION

A variety of loss functions are built into machine learning libraries such as Keras and Tensorflow. Our project involves outputting predictions of six different types of landcover types commonly found in Eastern North Carolina: water, developed, forest, cultivated, barren, and wetland. In traditional categorical crossentropy, the predicted labels, which come from the output of the softmax function, is compared to the ground truth label by the following function:

Categorical Crossentropy Loss Function

$$D(S, L) = -\sum_i L_i \cdot log(S_i)$$

After training our classifier, we examined how classes were mislabelled. We noticed that some classes were confused with other classes more often, as represented by this confusion matrix:



OBJECTIVES

The regular categorical crossentropy loss function only takes into account the degree by which the outputted and the correct answer vary, but does not look at patterns in incorrect answers.

The objective was to create a new crossentropy function that extends categorical crossentropy by introducing loss factors from the incorrect label data in the output. The hypothesis is that harnessing past data on which incorrect labels appear more often can speed up the learning process and place emphasis on differentiating labels that are often confused with each other.

METHODS

The Proposed Loss Function was built to dynamically modify after each loss *D* is calculated. The function modifies by changing the entries in the weighted cost matrix *A* after every loss calculation. The columns in the cost matrix correspond to entries in the label, and the entries in each column are weighted depending on how often each corresponding output entry appears and the weight of the corresponding output entry. The loss is therefore calculated not only by the magnitude of the correct label in the output, by also by the relative magnitudes of each of the incorrect predictions of the output as well.

Proposed Loss Function

$$D(S, L, A) = -\sum_i^n L_i \cdot log(S_i) + \sum_i^n [A \cdot L]_i \cdot E(S_i) + \sum_i^n L_i \cdot E(S_i)$$

$$A = Weighted\ Cost\ Matrix,\ A_i = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}$$

$$E(x) = e^{-3.2+3.3x} - .4728$$

MATRIX UPDATING

The loss D is calculated according to this equation and returned as the loss value to the neural network. After the loss is calculated, the matrix A is updated using the observed outputs S and the correct label L. The matrix is updated as follows:

1. Identify the column in matrix A to be updated – this is the column that corresponds to the correct label
2. Aggregate the predictions given to incorrect labels and find the average percentage given per label
3. Iterate through each entry in the output, ignoring the entry corresponding to the correct label
4. If an entry is larger than the measured average, raise the corresponding entry in matrix A, using a weighted sigmoid to maintain entry bounds
5. If an entry is lower than the measured average, lower the corresponding entry in matrix A, using a weighted sigmoid to maintain entry bounds
6. Return the new matrix A

The weighted cost matrix is multiplied in the loss function first by the label to isolate the correct column, and then by the exponential function of the predicted label S_i. By lowering or raising entries in the cost matrix, the loss function eventually builds in higher loss for classes that are commonly obfuscated with the correct class.
After each loss is calculated, the matrix column that corresponds to the correct label is updated to weight a higher loss for commonly obfuscated (and incorrect) labels.

CODE

```
def matrixBasedCrossentropy(output, target, matrixA, from_logits=False):
    Loss = 0
    ColumnVector = np.matmul(matrixA, target)
    for i, y in enumerate(output):
        Loss-= (target[i]*math.log(output[i],2))
        Loss+= ColumnVector[i]*exponential(output[i])
        Loss-= (target[i]*exponential(output[i]))
    newMatrix = updateMatrix(matrixA, target, output, 4)
    return [Loss, newMatrix]
```

```
def updateMatrix(MatrixA, Label, Output, S):
    Index = findLabelEntry(Label)
    ColumnOfInterest = np.matmul(MatrixA, Label)
    LenWrong = len(Output) - 1

    PercCorrect = np.dot(Label, Output)
    PercWrong = 1 - PercCorrect
    AvgX = PercWrong/LenWrong
    newColumn = np.zeros(len(Output))

    for ix, entry in enumerate(Output):
        x = InverseSigmoid(ColumnOfInterest[ix])
        Delta = abs(entry - AvgX)
        if(ix == Index):
            newColumn[ix] = ColumnOfInterest[ix]
            continue
        if(entry > AvgX):
            newX = ForwardSigmoid(x + Delta)
            newColumn[ix] = newX
        if(entry < AvgX):
            newX = ForwardSigmoid(x - Delta)
            newColumn[ix] = newX

    for x in range(len(MatrixA[0])):
        MatrixA[x][Index] = newColumn[x]
    print(np.asarray(MatrixA))
    print('\n')
    return(MatrixA)
```

OTHER LOSSES EXPLORED

Hinge Loss $\ell(y) = \max(0, 1 - t \cdot y)$

Logcosh $L(y, y^p) = \sum_{i=1}^{n} \log(\cosh(y_i^p - y_i))$

Huber Loss: $L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta\,|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$

Logcosh loss is twice differentiable everywhere, allowing for using Newton's method to find the optimum. By using the Hessian, boosting can be introduced to arrive at the optimum weights faster.

RESULTS

The proposed loss function was able to return higher loss in response to competing, commonly obfuscated data labels, and lowered loss when incorrect predictions were distributed more evenly across the different labels. However, after using this loss on the RCNN, there was little improvement in accuracy. Further modifications and tuning would need to be carried out to adjust the exponential function and sigmoid function parameters used to map data. Further modifications could also include adding boosting.

REFERENCES

https://www.machinecurve.com/index.php/2019/10/15/how-to-use-hinge-squared-hinge-loss-with-keras/

https://hearbeat.fritz.ai/boosting-your-machine-learning-models-using-xgboost-d2cabb3e948f

https://Towardsdatascience.com/custom-loss-functions-for-deep-learning-predicting-home-values-for-keras-for-r-532c9e098d1f

https://Keras.io/losses