

Imbalanced Multi-Label Classification using Convolutional Neural Networks

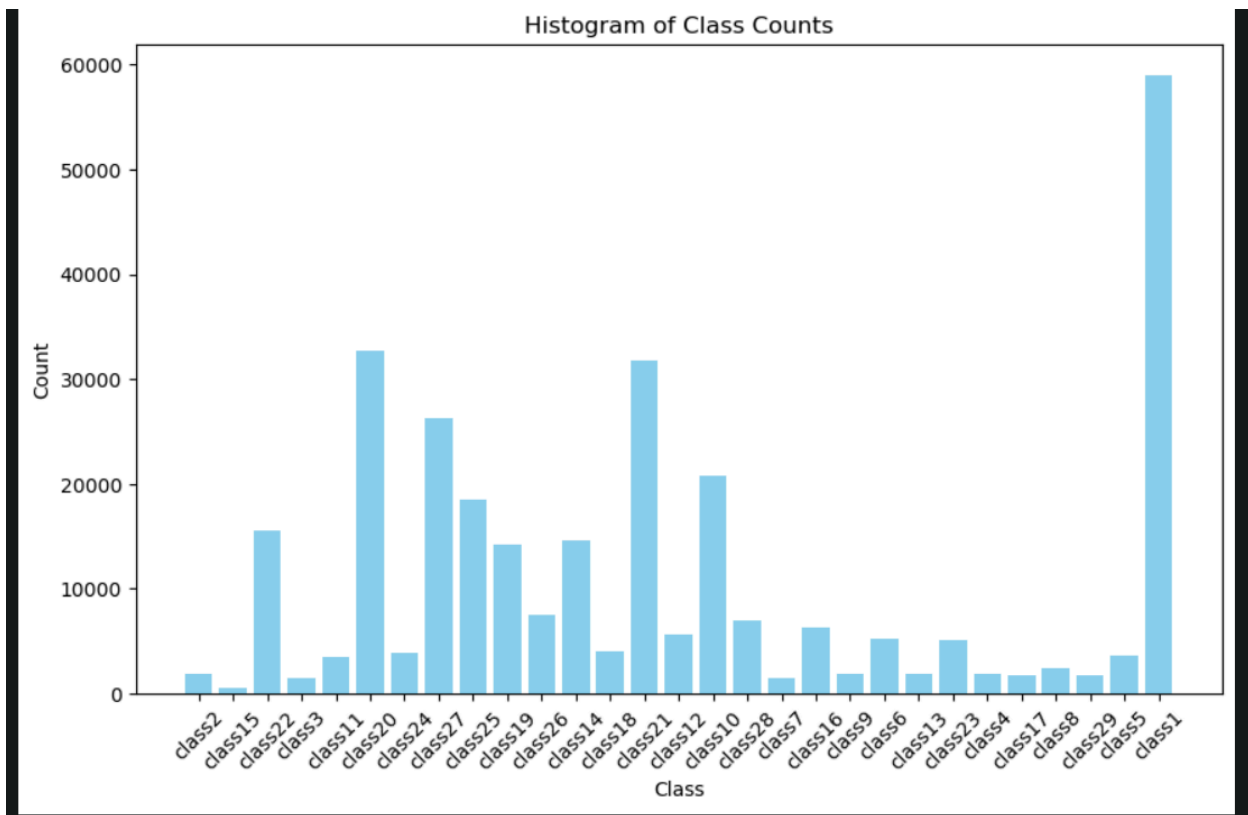
Instructor: Dr Amir Hossein Jafari

The exam focused on developing a deep neural network using Pytorch to classify images into different labels. The total number of classes in which we had to classify was 29. The primary metric I used for the evaluation was F1-micro and F1 macro. The best model had an F1 micro score of 0.459 on the test samples and 0.39 on the validation samples.

The dataset provided had an Excel sheet with columns as image_id, split type - (train and test), Target(class1 to class29), and target class which is the binarized conversion of the target very similar to one hot encoding.

The training dataset has 64674 images and the test dataset had 7186 images. I did all the preprocessing on the training dataset and tested my model on the test samples. The training dataset was highly imbalanced and images had different sizes like 1024 x 1024 x3, 475 x 350 x 3, etc. So resizing was performed. I resized all the images to 400 x 400 x 3.

[illegible]



Modelling -

The best model is a Convolutional Neural Network with 7 conv layers. Every conv2d layer is developed with a Batchnorm and a pooling layer. Some parameters like activation function, batch size, and learning rate were experimented with and optimized during the exam.

```

super(Conv, self).__init__()

self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=3)
self.convnorm1 = nn.BatchNorm2d(16)
self.pad1 = nn.ZeroPad2d(2)

self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=3)
self.convnorm2 = nn.BatchNorm2d(32)

self.conv3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=3)
self.convnorm3 = nn.BatchNorm2d(64)

self.conv4 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=3)
self.convnorm4 = nn.BatchNorm2d(128)

self.conv5 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=3)
self.convnorm5 = nn.BatchNorm2d(256)

self.conv6 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=3)
self.convnorm6 = nn.BatchNorm2d(512)

self.conv7 = nn.Conv2d(in_channels=512, out_channels=1024, kernel_size=3, padding=3)
self.convnorm7 = nn.BatchNorm2d(1024)

```

Initially, i used Maxpooling2d but the model wasn't performing well - so I replaced it with an average pooling 2d followed by a global average pool layer

```

self.pool = nn.AvgPool2d(kernel_size=2, stride=2)
self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))

self.fc1 = nn.Linear(in_features=1024, out_features=512)
self.fc2 = nn.Linear(in_features=512, OUTPUTS_a)
self.act = nn.ReLU()

```

Unit tests Explain Docstring

I experimented with LeakyRelu and Simple Relu and I finalized RElu for my modeling because it was giving better results.

The forward function has a convolutional layer - followed by an activation layer - followed by a batch norm layer and then an average pool 2d layer

Approaches that boosted my f1 score -

1. Employing a weighted loss function - i built a weighted BCEWithLogitLoss function inherited from nn.Module. The parameters which were passed were positive weights, negative weights, base logit function(Binary cross entropy)

```
class WeightedBCEWithLogitsLoss(nn.Module):
    """Unit tests Explain Docstring"""
    def __init__(self, pos_weights, neg_weights):
        super(WeightedBCEWithLogitsLoss, self).__init__()
        self.pos_weights = pos_weights
        self.neg_weights = neg_weights
        self.loss_fn = nn.BCEWithLogitsLoss(reduction='none')

    """Unit tests Explain Docstring"""
    def forward(self, logits, targets):
        # Compute BCEWithLogitsLoss for each label
        loss_per_label = self.loss_fn(logits, targets)

        # Apply weights based on the label
        # weighted_loss_per_label = torch.where(targets == 1, loss_per_label * self.pos_weights, loss_per_label)
        pos_loss = self.pos_weights * targets * loss_per_label

        # Apply negative weights where target is 0
        neg_loss = self.neg_weights * (1 - targets) * loss_per_label

        total_loss = pos_loss + neg_loss

        # Compute the mean loss across all labels
        loss = torch.mean(total_loss)

        return loss
```

Calculation of weights assigned to every label - Well it was the most difficult and tricky task to assign the correct weights(1 positive and 1 negative weight) based on the occurrence in the training dataset

```

class_counts_pos = {}
class_counts_neg = {}

# Create a set of all possible classes
all_classes = set()

for row in xdf_dset.target:
    classes = row.split(',')
    all_classes.update(classes)


# Initialize counts for all classes to 0
for class_name in all_classes:
    class_counts_pos[class_name] = 0

# Update counts for each class
for row in xdf_dset.target:
    classes = row.split(',')

    for class_name in all_classes:
        if class_name in classes:
            class_counts_pos[class_name] += 1

for class_name in all_classes:
    class_counts_neg[class_name] = len(xdf_dset) - class_counts_pos[class_name]

```

Usage  Unit tests Explain Docstring

```

def convert_counts_to_format(presence_counts, non_presence_counts):
    result = {}
    for class_name, presence_count in presence_counts.items():
        result[class_name] = {'0': non_presence_counts[class_name], '1': presence_count}
    return result

```

```

result = convert_counts_to_format(class_counts_pos, class_counts_neg)

class_weights = {}
positive_weights = {}
negative_weights = {}

for label, counts in result.items():
    num_1s = counts.get('1', 0)
    num_0s = counts.get('0', 0)
    total_samples = num_1s + num_0s

    positive_weights[label] = total_samples / (2 * num_1s) if num_1s != 0 else 0
    negative_weights[label] = total_samples / (2 * num_0s) if num_0s != 0 else 0

class_weights['positive_weights'] = positive_weights
class_weights['negative_weights'] = negative_weights

```

I used 2 loss functions in my modeling. While training i used a weighted loss function and while testing I used a simple BCEWithLogit Loss function to check how my model is generalizing on unseen data.

Data Augmentation, Clustering analysis and creating new Training data -

Clustering analysis - As i noticed that the data is highly imbalanced and we can't really augmented the images in the minority on fly as 1 image has multiple labels and it can lead to oversampling that label that is already in majority. So to deal with this problem i performed clustering analysis to cluster images that are similar and augment only those that have the less counts

To extract the features out of the images i used resnet feature extractor without the classification head

```
# Define a transformation to preprocess the images before passing them to the model
preprocess = transforms.Compose([
    transforms.Resize(400),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

Preprocessing on the images before clustering

```
# Iterate over the augmented images folder to extract features
for id_value in xdf_dset['id']:
    image_path = os.path.join(DATA_DIR, str(id_value)) # Construct full image path
    # print(image_path)
    image_paths.append(image_path)

    # Load and preprocess the image
    image = Image.open(image_path)
    image = image.convert("RGB")
    image = preprocess(image)
    image = image.unsqueeze(0) # Add batch dimension
    image = image.to(device)

    print(image)

    # Extract features using the pre-trained ResNet model
    with torch.no_grad():
        features = resnet_feature_extractor(image)

    # Convert features to numpy array and flatten
    features_np = features.cpu().numpy().flatten()
    image_features.append(features_np)
```

```
image_features_np = np.array(image_features)

print(image_features_np)

# Perform dimensionality reduction using PCA to reduce the feature dimensionality
pca = PCA(n_components=50) # You can adjust the number of components as needed
image_features_pca = pca.fit_transform(image_features_np)

# Apply K-means clustering to the extracted features
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
cluster_labels = kmeans.fit_predict(image_features_pca)

# Create a DataFrame to store the image paths and their corresponding cluster labels
cluster_df = pd.DataFrame({'Image_Path': image_paths, 'Cluster_Label': cluster_labels})

# Print the distribution of images in each cluster
print(cluster_df['Cluster_Label'].value_counts())

img_dir_1 = '/home/ubuntu/final_dl/Exam2-v5/excel/'

new_excel_file = os.path.join(img_dir_1, 'updated_cluster.xlsx')
cluster_df.to_excel(new_excel_file, index=False)
```



```
In [9]: dfc.head()
```

```
Out[9]:
```

	Image_Path	Cluster_Label
0	/home/ubuntu/final_dl/Exam2-v5/Data/img_234267...	1
1	/home/ubuntu/final_dl/Exam2-v5/Data/img_233874...	1
2	/home/ubuntu/final_dl/Exam2-v5/Data/img_234829...	1
3	/home/ubuntu/final_dl/Exam2-v5/Data/img_237639...	3
4	/home/ubuntu/final_dl/Exam2-v5/Data/img_239672...	1

```
In [10]: dfc.Cluster_Label.value_counts()
```

```
Out[10]: Cluster_Label
```

```
0    19118
1    16265
3    14019
4     8651
2     6621
Name: count, dtype: int64
```

This is the result of my clustering analysis - after this ill augment the images present in cluster labels 4 and 2

Data Augmentation -

I experimented with alot of data augmentation techniques like applying adaptive blur, coarse dropout, channel dropout, random brightness - but i was getting black augmented images with mostly 0's in the output so after alot of trial and error i finalised the following

```

# Define augmentation pipeline
aug_pipeline = iaa.Sequential([
    iaa.Fliplr(0.7),
    iaa.AverageBlur(k=(2, 7)),
    iaa.MedianBlur(k=(3, 11)),
    iaa.GaussianBlur(sigma=(0.0, 3.0)),
    iaa.PerspectiveTransform(scale=(0.01, 0.15)),
    iaa.ElasticTransformation(alpha=(0, 3.0), sigma=0.25),
    iaa.CropAndPad(percent=(-0.25, 0.25)),
    iaa.Cutout(nb_iterations=(1, 3), size=0.2, squared=False)

    # Add more augmentation techniques as needed
])

# to_tensor = transforms.ToTensor()

```

```

# Get the count of cluster 0
cluster_0_count = cluster_df['Cluster_Label'].value_counts()[0]

print(cluster_0_count)

# Get the count of each cluster other than 0
other_clusters_count = cluster_df['Cluster_Label'].value_counts()[3:]

print(other_clusters_count)

# Calculate the count difference between each cluster and cluster 0
count_diff = cluster_0_count - other_clusters_count

count_diff = count_diff.to_dict()

print(count_diff)

# to_tensor = transforms.ToTensor()

```

Post augmentation results

```
Cluster_Label
4      19118
2      19118
0      19118
1      16265
3      14019
Name: count, dtype: int64
```

Mapping the target and target class for augmented images

-> After augmenting the images and increasing the count of images belonging to minority labels - i had to map those images with their target labels - which would be same as the labels for there original images

```
df3.Cluster_Label.value_counts().sum()

87638
```

I applied data augmentation and created ~23,000 more images and appended those into the excel for the train split

Thresholding - The best threshold that worked for me was 0.2.

It was a great competition and I learned a lot about image augmentation, building a strong cross-validation, class imbalance problem and developing a strong DL model.

