Multi-Label Classification on the PASCAL VOC Dataset

Joel Huang
1002530

Daniel Chin
1002095

{joel_huang, daniel_chin}@mymail.sutd.edu.sg

***Abstract*— This report contains the procedure, results and evaluation of the models used in the mini project for 50.039 - Theory and Practice of Deep Learning. The main task involves transfer learning for pretrained models to perform multi-label classification on the PASCAL VOC 2012 dataset.**

I. INTRODUCTION

A. Defining Multi-label Classification

We distinguish the multi-label classification problem from the similar sounding, but different multi-class classification problem. Multi-class classification is a supervised learning problem with one ground truth class label per sample. Multi-label classification is also a supervised learning problem, but each sample can contain multiple classes. The goal is to predict, for each input, some label at some confidence value for every class present in the dataset [1]. Further, this can be thresholded to obtain direct classification scores.

B. Formulation

Formally, the classification goal is to predict a label in the output space $\mathcal{Y}$, given samples $x \in \mathcal{X}$:

$$f : \mathcal{X} \to \mathcal{Y} \in \{0, 1\} \tag{1}$$

This multi-label classifier essentially produces for every class c, a binary classifier with probabilities $p_c(x_i) = P(Y = 1 \,|\, X = x_i)$, and $1 - p_c(x_i) = P(Y = 0 \,|\, X = x_i)$ for all samples $x_i \in X$.

The output probability $p_c(x_i)$ can be converted to a classification with the use of a threshold t, where the classification prediction $Y = 1$ is assigned to $p_c(x_i) \geq t$ and $Y = 0$ is assigned to $p_c(x_i) < t$.

II. CUSTOM DATASET AND DATALOADER CLASSES

The PASCAL VOC 2012 dataset[2] consists of images with multiple labels for each object in the image. There are a total of 20 classes $c \in C$: *aeroplane, bicycle, bird, boat, bottle, bus, car, cat, chair, cow, diningtable, dog, horse, motorbike, person, pottedplant, sheep, sofa, train,* and *tvmonitor*. See Fig. 2 for a overview of the class distributions. We note that there are text files with the training and validation splits, in addition to the raw annotations given in `.xml` files. We parse both to obtain both single and multi-instance labels for each image.

A. Single and multiple instances

Since this is a multi-label dataset, images can have any number of labels for any of the 20 classes. We generate single instance labels in order to fulfill our binary classification task, as each classifier requires a single ground truth per image. In this way, we obtain from the training set a total of 5717 images with a total of 8863 labels, and in the validation set a total of 5823 images with a total of 8856 labels. As seen in Fig. 1, multiple instances of a class are present. We collapse them into a single instance, such that $Y = \{bird_1, \cdots, bird_{11}\}$ becomes the one-hot vector $Y = [Y_0 = 0, Y_1 = 0, Y_2 = 1, Y_3 = 0, \cdots, Y_{19} = 0]$ (The label for `bird` corresponds to Y_2).

Fig. 1: 11 instances of `bird` in a single image.

B. Constructor

The class constructor takes the directory path containing the images, the split type (`'train'` or `'val'`), and the transforms to be applied to the dataset. It then loads all the data paths without explicitly loading any images. However, string labels are loaded.

```
class VOCDataset(Dataset):
  def __init__(self, directory, split, transforms=
    None, multi_instance=False, verbose=False):
    self.split = split
    self.verbose = verbose
    self.directory = directory
    self.transforms = transforms
    self.multi_instance = multi_instance
    self.labels_dict = self.get_labels_dict()
```

```
    self.label_count, self.data = self.
    _load_all_image_paths_labels(split)
    self.classes_count = self._count_classes()
```

C. Subclass implementations

We need to necessarily implement `__len__()` and `__getitem__()` since we are subclassing `torch.utils.data.Dataset`. To be more efficient, we only load images when the item is accessed via `dataset[index]`.

```
def __len__(self):
  return len(self.data)

def __getitem__(self, idx):
  image = self._load_image(self.data[idx]['
  image_path'])
  if self.transforms is not None:
    image = self.transforms(image)
  labels = self.data[idx]['labels']
  return (image, labels)
```

D. Loading data

Three private methods for interfacing with the `.xml` files. The first helper function scrapes for the correct `.xml` file, while the other two retrieve image paths and labels with multi instance support if necessary.

```
def _get_xml_file_path(self, image_name):
  xml_name = image_name + '.xml'
  xml_path = os.path.join(self.directory, '
  Annotations', xml_name)
  return xml_path

def _load_all_image_paths_labels(self, split):
  label_count = 0
  all_image_paths_labels = []
  images_list = self._get_images_list(split)
  xml_path_list = [self._get_xml_file_path(
  image_path)
          for image_path in images_list]
  for image_path, xml_path in zip(images_list,
  xml_path_list):
    image_path = os.path.join(self.directory, '
  JPEGImages', image_path + '.jpg')
    assert(image_path not in
  all_image_paths_labels)
    if self.multi_instance:
      labels = self._get_labels_from_xml(xml_path
  )
    else:
      labels = list(np.unique(self.
  _get_labels_from_xml(xml_path)))
    label_count += len(labels)
    if self.verbose:
      print("Loading labels of size {} for {}..."
  .format(
        len(labels), image_path))
    image_path_labels = {'image_path': image_path
  ,
                'labels': labels}
    all_image_paths_labels.append(
  image_path_labels)

  print("SET: {} | TOTAL IMAGES: {}".format(self.
  split, len(all_image_paths_labels)))
  print("SET: {} | TOTAL LABELS: {}".format(self.
  split, label_count))
  return label_count, all_image_paths_labels

def _get_labels_from_xml(self, xml_path):
  labels = []
  tree = ET.parse(xml_path)
  root = tree.getroot()
  for child in root.iter('object'):
    labels.append(child.find('name').text)
  return labels
```

Two private methods here for image loading. The first method `_load_image()` loads a 3-channel image as a `PIL Image`. The second method `_get_images_list()` returns a list of image paths. We load files from paths in `ImageSets/Main/train.txt` and `ImageSets/Main/val.txt`.

```
def _load_image(self, image_path):
  img = Image.open(image_path)
  assert(img.mode == 'RGB')
  return img

def _get_images_list(self, split):
  image_paths = []
  image_path_file = os.path.join(self.directory,
  'ImageSets/Main', split + '.txt')
  with open(image_path_file) as f:
    for image_path in f.readlines():
      candidate_path = image_path.split(' ')[0].
  strip('\n')
      image_paths.append(candidate_path)
  return image_paths
```

E. Batching for encoded labels

We also implement a `Batch` class to help us to return one-hot encoded labels, which are useful for tensor conversions and comparisons for the multi-label binary classification task.

```
class VOCBatch:
  def __init__(self, data):
    self.transposed_data = list(zip(*data))
    self.image = torch.stack(self.transposed_data
    [0], 0)
    self.labels = self.construct_int_labels()

  def construct_int_labels(self):
    remap_dict = VOCDataset.get_labels_dict(None)
    labels = self.transposed_data[1]
    batch_size = self.image.shape[0]
    num_classes = len(remap_dict)
    one_hot_int_labels = torch.zeros((batch_size,
    num_classes))
    for i in range(len(labels)):
      sample_labels = labels[i]
      one_hot = torch.zeros(num_classes)
      sample_int_labels = []
      for string_label in sample_labels:
        int_label = remap_dict[string_label]
        one_hot[int_label] = 1.
      one_hot_int_labels[i] = one_hot
    return one_hot_int_labels

  def pin_memory(self):
    self.image = self.image.pin_memory()
    self.labels = self.labels.pin_memory()
    return self

def collate_wrapper(batch):
return VOCBatch(batch)
```

III. MODEL AND METRICS

A. Model

For the transfer learning task, we choose a 34-layer ResNet[3] for its ensemble-like behavior and widely available pre-trained weights. We initialize the model with pre-trained weights from PyTorch trained on ImageNet and train

all trainable parameters in the model. This allows us to converge more quickly.

B. Average precision

Although used in classification problems like multi-class classification, accuracy is not a good metric for multi-label classification. In the multi-label classification problem, we note that predictions can be fully correct, partially correct, and fully incorrect[1]. If we look at

$$\text{Accuracy} = \frac{TP + TN}{P + N}, \tag{2}$$

we can see that a multi-label prediction scored on this metric will have no meaning. Consider a toy example where the true labels are $Y = [\,1, 0, 0, 0, 0\,]$. If our network produces a prediction $\hat{Y} = [\,0, 0, 1, 0, 0\,]$, we would achieve an accuracy score of 0.6: we are assigning more than due credit to our TN predictions. An alternative is to use Average Precision (AP).

$$\text{AP} = \sum_n (R_n - R_{n-1}) P_n \tag{3}$$

According to the implementation notes from scikit-learn [4], AP summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight. Here, P_n and R_n are the precision and recall at the n-th threshold.

Going back to our toy example, if our prediction were to be evaluated instead on AP, we would obtain a AP of:

$$\sum_{n \in \{0, \cdots, 1.0\}} (R_n - R_{n-1}) P_n = 0.2 \tag{4}$$

C. Tail accuracy

The Tail Accuracy (Tailacc) at a specified threshold value is the precision of where the threshold is used to determine if the class prediction is positive or negative. This was done by computing the Tailacc(t) class-wise over 10 intervals beginning at 0.5 and ending at 0.95. The average Tailacc was then computed for each threshold value. The equations below show how we compute the score.

$$TP = \sum_{i=1}^{m} I[f(x_i) = y_i] I[f(x_i) > t] \tag{5}$$

$$FP = \sum_{i=1}^{m} I[f(x_i) \neq y_i] I[f(x_i) > t] \tag{6}$$

$$Tailacc(t) = \frac{TP}{TP + FP} \tag{7}$$

IV. LOSS FUNCTION

A. Loss weighting

We plot the distribution of the training set in Fig. 2. The blue bars represent the frequency of the single instance labels in the training set. This means at training time, we see samples according to this frequency. Consequently, our classifiers also learn from samples according to this distribution. To account for the class imbalance and ensure our classifiers are equally trained, we enforce a class-wise loss weighting $W = \{w_0, \cdots, w_{19}\}$ proportional to the frequency of the class c in question:

$$w_c = \frac{\min_k count(k)}{count(c)}, \tag{8}$$

where $c, k \in C$. Each $0 \leq w_c \leq 1$ is therefore the scaling factor of the class-wise loss at training time. We can confirm their levelling effect by plotting the factors as a percentage of the total occurrences (the red bars in Fig. 2).

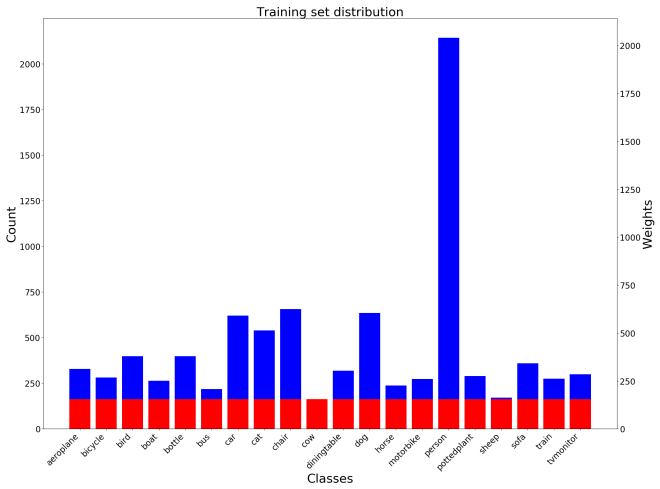

Fig. 2: (a) Blue: Distribution of single-instance class labels in the training set, (b) Red: Corrected distribution used to weight loss function.

B. Binary Cross-Entropy Loss

We choose a binary cross-entropy (BCE) loss to optimize our 20 binary classifiers. BCE loss is derived from the idea of maximum likelihood and is closely related to the Kullback-Leibler divergence of two probability distributions. The total loss function for one mini-batch with size N is given below, where $\hat{Y}$ and Y are tensors with shape (N, C):

$$\frac{1}{N} \sum_{c=0}^{C} \mathcal{L}(\sigma(\hat{Y}), Y), \tag{9}$$

the average of N point losses summed over all classes $c \in C$, where the loss function $\mathcal{L}$ is

$$\mathcal{L}(\sigma(\hat{y}_n), y_n) = -y_n \cdot \log(\sigma(\hat{y}_n)) - (1 - y_n) \cdot \log(\sigma(-\hat{y}_n)) \tag{10}$$

and σ is the logistic function $\sigma(\hat{y}) = \dfrac{1}{1 + e^{-\hat{y}}}$. The `BCELossWithLogits` implementation in PyTorch combines the sigmoid function with binary cross-entropy loss in a numerically stable way. The output loss for the mini-batch $\mathcal{L}_b$ is a thus a tensor of size $(N,)$.

V. RESULTS

A. Training results

As training is expensive, we truncate our results to 50 epochs. The averaged training and validation losses over the first 50 epochs can be found in Fig. 3.

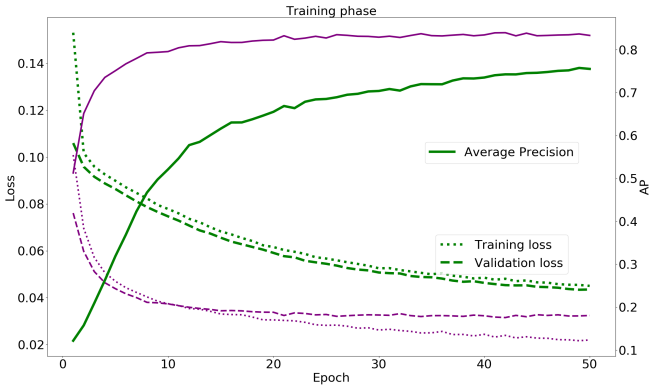

Fig. 3: (a) Green curve: $lr = 10^{-3}$ (b) Purple curve: $lr = 10^{-2}$

For ResNet34 with learning rate 10^{-3}, there is little evidence of overfitting throughout the training process. The training loss converges, but we expect room for improvement if further training is carried out after 50 epochs. Training loss remains marginally higher than validation loss for all epochs, which might be indicative of further improvement in model fitting.

For ResNet34 with learning rate 10^{-2}, we can see the sharp drop within the first few epochs compared to the green curve. However, training loss starts to decrease and diverge from validation loss after 20 epochs, which could indicate some over-training. The validation loss for the purple curve is largely unchanging after 20 epochs.

B. Tail accuracy

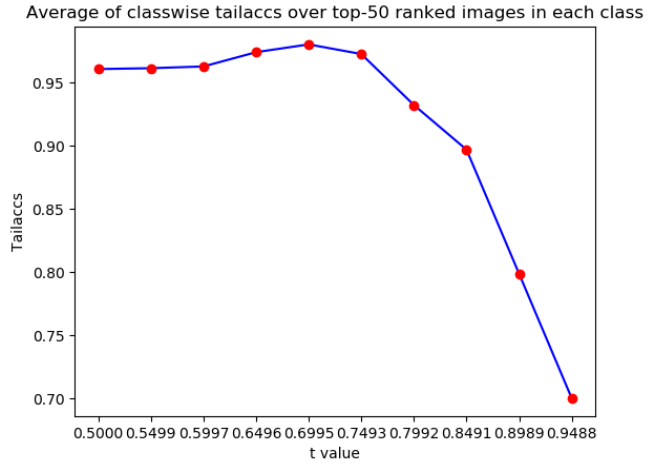

Fig. 4: ResNet34 $lr = 10^{-3}$: Tail accuracy increasing, then falling off as t-value increases.

In Fig. 4 we plot the tail accuracy for $lr = 10^{-3}$. We see the tail accuracy increasing, then falling off. The rising tail accuracy as the threshold value rises can be explained by more samples being categorized as negative and therefore being excluded from the count used to compute the tail accuracy. The fall off after $t = 0.75$ is due to the drop in true positive count as compared while the count of false positive is relatively unchanged. The flatline observed in Fig. 5 is

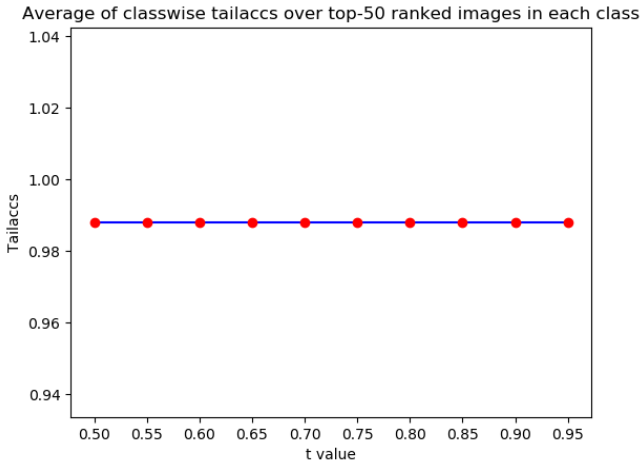

Fig. 5: ResNet34 $lr = 10^{-2}$: Tail accuracy flatlining as t-value increases.

quite coincidental. The outputs of the top-50 ranked images in the each class are well above 0.95. However, for some of the classes there are small amounts of false positives, on average 1 per class. This resulted the observed average tail accuracy results.

VI. EVALUATING IMAGE AUGMENTATIONS

Our basic `transforms` stack includes a `RandomResizedCrop(300)`, `ToTensor`, and `Normalize(mean, std)`. These are applied to both the training set and validation set for all tasks. However, to ensure standardized, comparative performance of our models, we choose not to apply the additional augmentations, like flipping and rotation, on the validation set, to retain the true distribution of the original dataset. We acknowledge this is an active area of research.

We have computed a mean of $[0.4589, 0.4355, 0.4032]$ and standard deviation of $[0.2239, 0.2186, 0.2206]$. The reasons for choosing random cropping despite the potential loss of features are discussed below. By inspection, most of the images in the dataset have image heights and widths within 300 to 500 pixels. With this knowledge, we select `RandomResizedCrop` over `FiveCrop` and `CenterCrop` for a few reasons. Firstly, `FiveCrop` multiplies the batch size by 5. We avoid this due to VRAM constraints. Secondly, we avoid having to deal with mislabelled images. Because this is a multi-label classification task, there is a possibility of some of the five crops not containing the features required for correct classification. Additionally, we set the crop size to 300 pixels, so for a 300×500 image there is a good chance of our crop containing the features required, compared to `CenterCrop`, where features outside the center boundary are always discarded, even across epochs.

Note: as a separate experiment with image augmentations, we first tried staging them on a ResNet18, with $lr = 10^{-3}$. After cropping, we trained with three different sets of image augmentations separately: `RandomHorizontalFlip`, and `RandomHorizontalFlip` com-

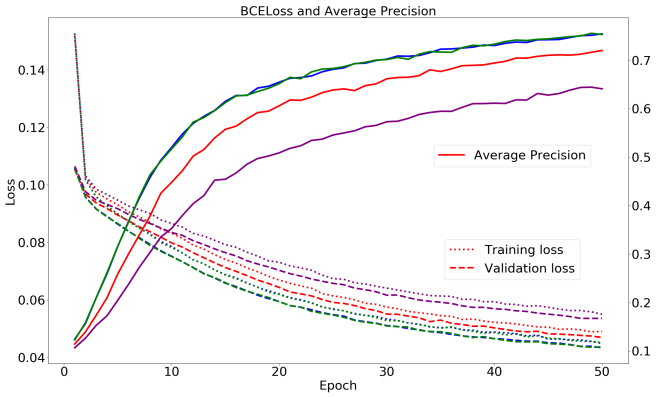

Fig. 6: Image augmentation validation and AP results for an alternative model, ResNet18, trained with $lr = 10^{-3}$.
(a) Green curve: No augmentation,
(b) Blue curve: RandomHorizontalFlip,
(c) Red curve: RandomHorizontalFlip + RandomRotation,
(d) Purple curve: RandomHorizontalFlip + RandomVerticalFlip + RandomRotation

bined with `RandomRotation`, and `RandomHorizontalFlip` combined with `RandomVerticalFlip`, and `RandomRotation` . The results are plotted in Fig. 6. It is surprising that adding more image augmentations actually reduces performance. In particular, adding `RandomRotation` decreases performance significantly, both in terms of AP and validation loss. We next look at image augmentation's effect on performance on ResNet34. We take our best ResNet34 model (with $lr = 10^{-2}$) from the training phase (see Fig. 3), and augment it with two separate methods, `RandomHorizontalFlip`, and `RandomRotation`. We see a very marginal improvement in AP after introducing random horizontal flips, but a hit to performance in both AP and training/validation loss with rotations. The results in Fig. 7 seem to reinforce the suggestion that rotations are not a good augmentation for this dataset and combination of task and loss function.

Fig. 7: Image augmentation validation and AP results for ResNet34 trained with $lr = 10^{-2}$.
(a) Green curve: No augmentation,
(b) Blue curve: RandomHorizontalFlip,
(c) Red curve: RandomRotation

VII. REPRODUCIBILITY

We release our training hyperparameters here: `https://goo.gl/rwWsqj`.

A. Hyperparameters

During model training, to ensure reproducibility, we fix the random seed with `torch.manual_seed(0)`. This ensures that all shuffles, random flips, etc. done using values obtained from the pseudo-random generator will be fixed and therefore anyone running training with the same setting should be able to obtain the same results.

B. Environment Setup

We use PyTorch v1.0.0 and Torchvision v0.2.2, with NumPy v1.14.5 and Scikit-learn v0.20.3. In addition, we used v8.6.8 for Tkinter GUI development.

VIII. GRAPHICAL DEMONSTRATOR

A simple GUI was implemented with Tkinter. The application supports prediction for a single image by selecting an image file from the local machine file system, and the browsing of top-ranked images for each class.

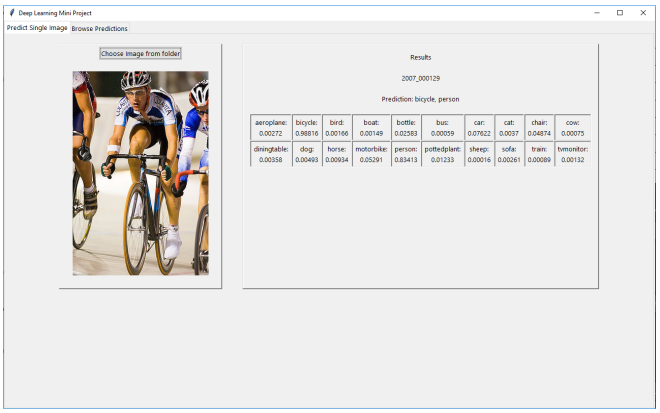

Fig. 8: Screenshot of prediction output for single image

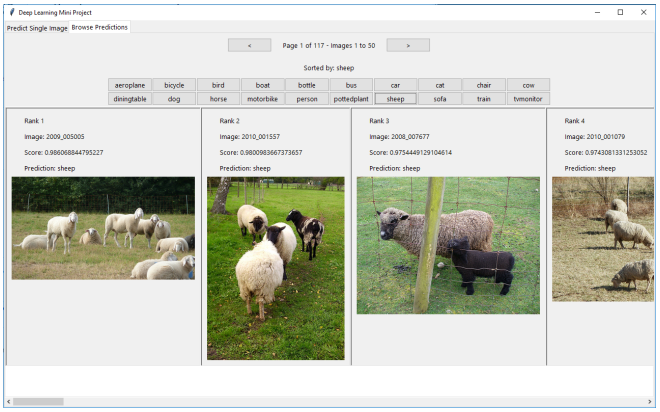

Fig. 9: Screenshot of browsing page for top-ranked sheep images

REFERENCES

[1] M. S. Sorower, "A literature survey on algorithms for multi-label learning," 2010.

[2] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results." http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[4] "Scikit-learn documentation, average_precision_score." https://scikit-learn.org/stable/modules/generated/sklearn.metrics.average_precision_score.html. scikit-learn v0.20.3.