ME592 Homework 3

Kojo Adu-Gyamfi, Md Zahid Hasan, Olivia Devitt, Sanghyoup Gu

Ⅰ. Introduction

In the “Berkeley Deep Drive” dataset, various measurements from an autonomous vehicle are provided including videos, images, GPS data, and so on. The goal of this assignment was to build and train a model given a labeled training set, testing set, and validation set. The model chosen was a convolutional neural network, which is a deep learning method useful for vision-based navigation.

Ⅱ. Image Label

In order to conduct heterogeneous multitask learning, the dataset provides labels for each image, which involves a variety of information such as weather and scene type.

Image

Label Structure

Label name	Sub label	Value	Description
attributes	scene	city street	Image type
	time of day	day time	-
	weather	clear	-
Labels (Objects)	attributes	Occluded, truncated	-
	box2d	x1, x2, y1, y2	Position
	category	traffic light	-
	id	0	Index for the object
	:	:	:
name	-	XXXX.jpg	Image name
timestamp	-	10000	The time image was captured

Table 1. Example image and label

From this project, weather labels are used for the classification of weather based on the input images.

Ⅲ. Image pre-processing

1. Training Data selection

The BDD dataset contains 10,000 instances for train, validation, and test purposes. The training dataset has 70,000 instances. The distribution of data is imbalanced, meaning some of the conditions were disproportionately represented in the dataset. In order to mitigate the effect of the imbalanced training data distribution, an equal number of images are used for each class. An exception is made for "foggy", because the "foggy" class has very few instances. For each class, 4,800 instances are used for training data, while all foggy instances are used.

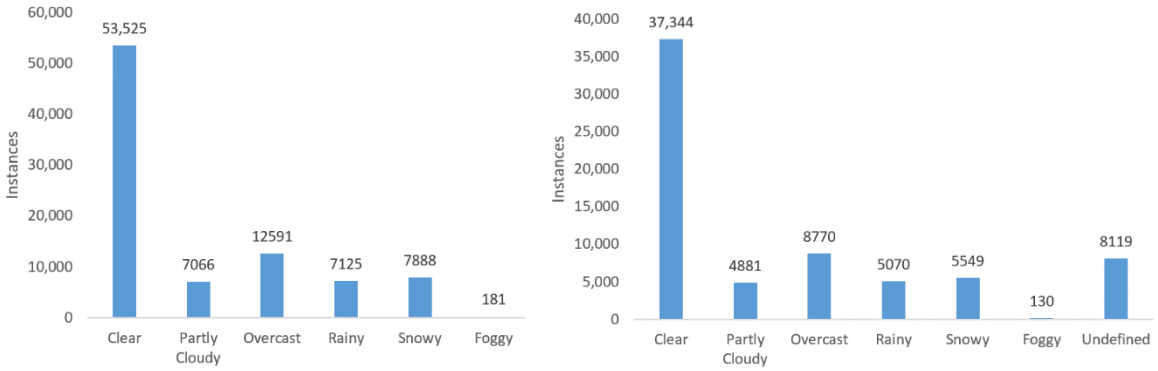


Figure 1. The number of instances of total and train dataset

Weather	Clear	Partly Cloudy	Overcast	Rainy	Snowy	Foggy	Undefined
# of instances used in training	4,800	4,800	4,800	4,800	4,800	130	4,800

Table 2. The number of instances used in train

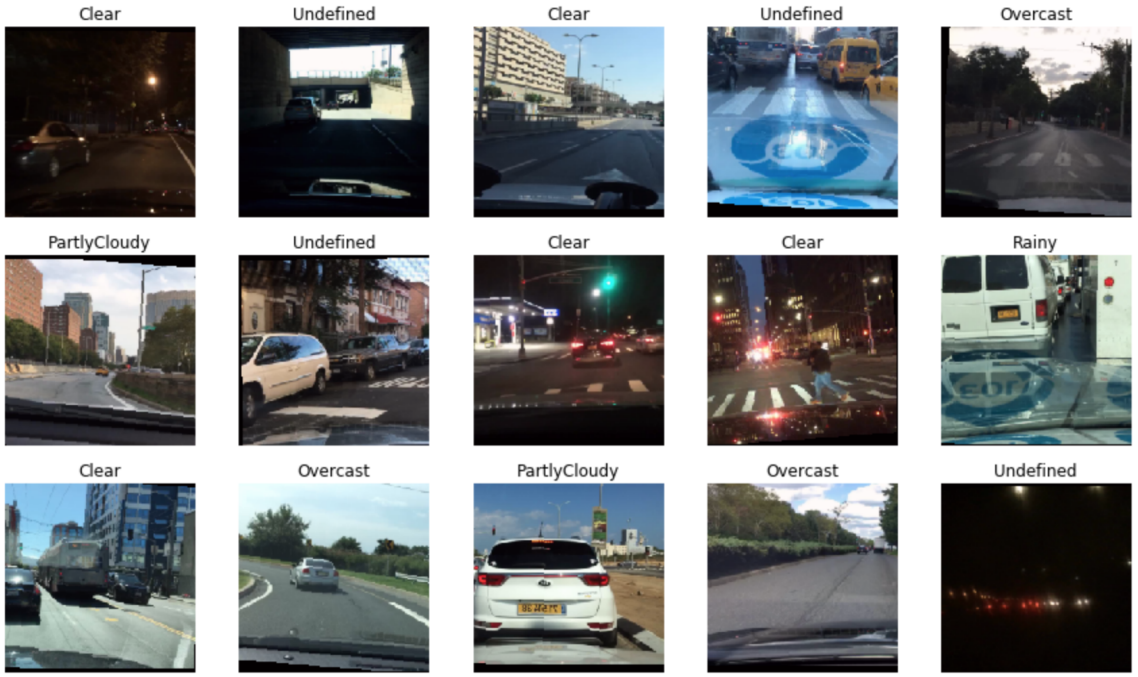
2. Image pre-processing

We start by setting a TRAIN RATIO, which determines what proportion of each class's images will be utilized in the training set, while the rest will be used in the test set. Within the data folder, we establish a train and test folder after eliminating any that are currently present. After that, we get a list of all classes and loop through them. We get the image names for each class, use the first TRAIN RATIO for the training set, and the rest for the test set. The photos are then copied into their corresponding train or test folders using shutil.copyfile.

Our train/test splits are now complete. To normalize our dataset, we compute its mean and standard deviation (std). With dim = (1,2), we loop through each image and calculate the mean and standard deviation across the height and width dimensions, totaling all the means and stds, and then dividing them by the number of samples, len(train data), to determine the average. The original images are 120 x 720 pixels, and because we'll be utilizing a pre-trained model (RESNET), we'll need to make sure our images are the same size and normalized as the ones used to train the model. We employ the same data augmentation techniques as before: random rotation, horizontal flipping, and

cropping. All pre-trained models require input images that are normalized in the same way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W), with H and W of at least 224. The images were imported into a range of [0, 1] and then normalized using the train data's mean and standard deviation.

```
pretrained_size = 224
pretrained_means = [0.2786, 0.2925, 0.2898]
pretrained_stds= [0.1975, 0.1981, 0.2008]

train_transforms = transforms.Compose([
                           transforms.Resize(pretrained_size),
                           transforms.RandomRotation(5),
                           transforms.RandomHorizontalFlip(0.5),
                           transforms.RandomCrop(pretrained_size, padding = 10),
                           transforms.ToTensor(),
                           transforms.Normalize(mean = pretrained_means,
                                                std = pretrained_stds)
                       ])

test_transforms = transforms.Compose([
                           transforms.Resize(pretrained_size),
                           transforms.CenterCrop(pretrained_size),
                           transforms.ToTensor(),
                           transforms.Normalize(mean = pretrained_means,
                                                std = pretrained_stds)
                       ])
```



Ⅳ. Model Structure

We'll be implementing one of the ResNet (Residual Network) model versions in this assignment (ResNet50). ResNet, like VGG, includes a variety of setups that specify the number of layers and their sizes. Each layer is built up of convolutional layers, batch normalization layers, and residual connections, which are all made up of blocks (also called skip connections or shortcut connections).

Why did we go with ResNets in the first place? The residual connections are the key. The gradient signal either explodes (becomes very huge) or vanishes (becomes very little) when it backpropagates through several layers, making training really deep neural networks problematic.

We'll also use a learning rate scheduler, which is a PyTorch wrapper around an optimizer that allows us to change the learning rate of the optimizer dynamically during training. We'll utilize the one-cycle learning rate scheduler from this work, which is also known as superconvergence and is widely used in the fast.ai course.

BasicBlock

The BasicBlock of the ResNet used is made of two 3x3 convolutional layers. The first, conv1, has a stride which varies depending on the layer (one in the first layer and two in the other layers), whilst the second, conv2, always has a stride of one. Each of the layers has a padding of one - this means before the filters are applied to the input image we add a single pixel, that is zero in every channel, around the entire image. Each convolutional layer is followed by a ReLU activation function and batch normalization. Below is the basic block .

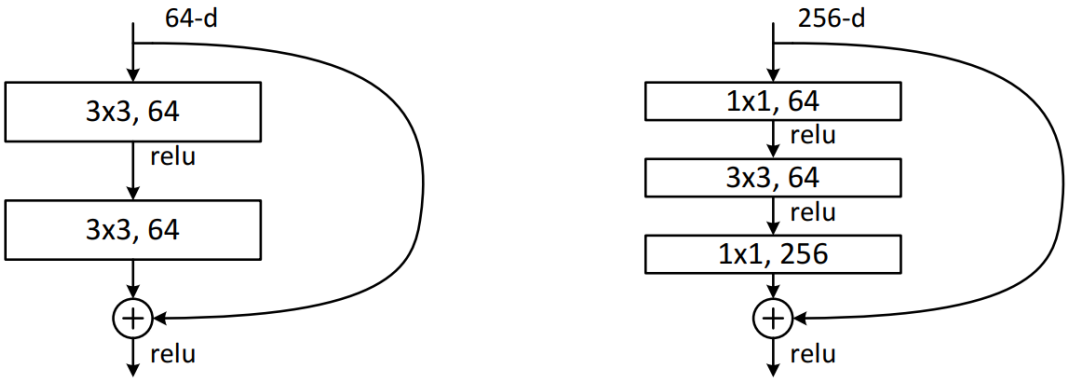
```
class BasicBlock(nn.Module):

    expansion = 1

    def __init__(self, in_channels, out_channels, stride = 1, downsample = False):
        super().__init__()

        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size = 3,
                               stride = stride, padding = 1, bias = False)
        self.bn1 = nn.BatchNorm2d(out_channels)

        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size = 3,
                               stride = 1, padding = 1, bias = False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.relu = nn.ReLU(inplace = True)

        if downsample:
            conv = nn.Conv2d(in_channels, out_channels, kernel_size = 1,
                             stride = stride, bias = False)
            bn = nn.BatchNorm2d(out_channels)
            downsample = nn.Sequential(conv, bn)
        else:
            downsample = None

        self.downsample = downsample
```

```
        def forward(self, x):

            i = x

            x = self.conv1(x)
            x = self.bn1(x)
            x = self.relu(x)

            x = self.conv2(x)
            x = self.bn2(x)

            if self.downsample is not None:
                i = self.downsample(i)

            x += i
            x = self.relu(x)

            return x
```

ResNet18, ResNet34, ResNet50, ResNet101, and ResNet152 are the different ResNet configurations based on the total number of layers within them.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

The table above shows that the first block of ResNet18 and ResNet34 has two 3x3 convolutional layers with 64 filters, with ResNet18 having two and ResNet34 having three of these blocks in the first layer. Bottleneck blocks are ResNet50, ResNet101, and ResNet152 blocks that have a different structure than ResNet18 and ResNet34 blocks. Bottleneck blocks constrict the amount of channels in the input before extending them again. ResNet18 and ResNet34 employ a conventional BasicBlock (left), while ResNet50, ResNet101, and ResNet152 use the Bottleneck block.



Because ResNet50 was pre-trained on the ImageNet dataset, which has 1000 classes, the final linear layer for classification has a 1000-dimensional output. However, our data only has a classification of 7, therefore we must first generate a new linear layer with the appropriate dimensions. After that, we replace the linear layer of the pre-trained model with our own, randomly initialized linear layer. The torchvision-provided pre-trained ResNet model does not provide an intermediate output, which we'd like to use for analysis. This is solved by creating our own ResNet50 model and then copying the pre-trained parameters into it.

Training the Model

To discover a good learning rate for our model, we'll use the learning rate finder. We begin by setting up an optimizer with a modest learning rate, designing a loss function (criterion), and putting the model and the loss function on the device. The center of the steepest descending slope would be a reasonable learning rate to use here.

The learning rates of our model can then be set using discriminative fine-tuning, a transfer learning strategy in which later layers in a model have higher learning rates than earlier layers.

The learning rate discovered by the learning rate finder was utilized as the maximum learning rate - which was used in the final layer - while the remaining layers gradually reduced the learning rate, gradually decreasing towards the input.

The learning rate scheduler was then configured. While the model is training, a learning rate scheduler dynamically changes the learning rate. The one-cycle learning rate scheduler will be used.

The one-cycle learning rate scheduler starts with a low starting learning rate and progressively increases it to a maximum value - the value determined by our learning rate finder - before gradually decreasing it to a final value lower than the beginning learning rate. After each parameter update step, i.e. after each training batch, this learning rate is adjusted. The learning rate for the final fc layer in our model.

To set-up the one cycle learning rate scheduler we need the total number of steps that will occur during training. We simply get this by multiplying the number of epochs with the number of batches in the training iterator, i.e. number of parameter updates. We get the maximum learning rate for each parameter group and pass this to max_lr.

Another thing we did was integrate top-k accuracy. Our job is to categorize an image into one of seven weather attribute classes; unfortunately, several of these classes appear to be very similar, making it difficult for a human to distinguish between them. So, when it comes to calculating

accuracy, perhaps we should be a little more liberal.

One method of solving this is using top-k accuracy, where the prediction is labeled correct if the correct label is in the top-k predictions, instead of just being the first. Our calculate_topk_accuracy function calculates the top-1 accuracy as well as the top-k accuracy, with k=5 by default.

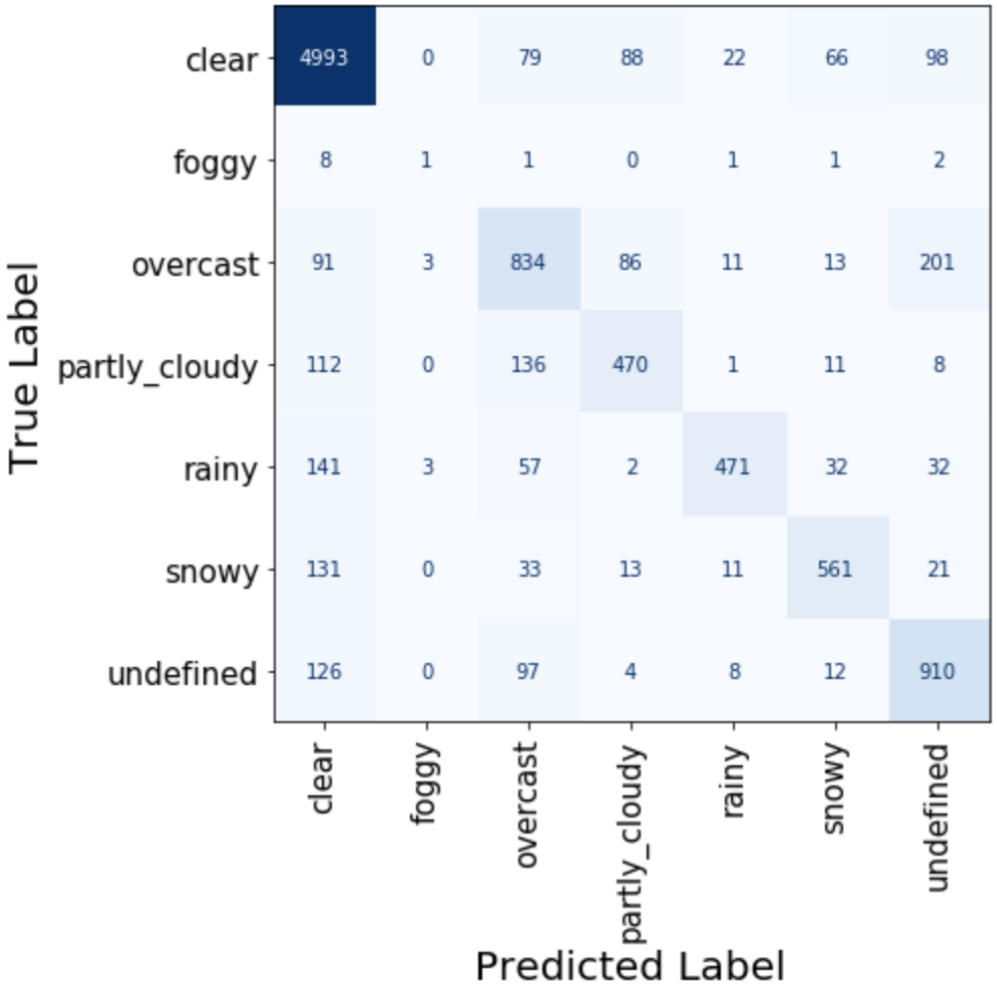
Finally, we can train our model and below are the actual results.

```
Epoch: 01 | Epoch Time: 33m 54s
        Train Loss: 0.747 | Train Acc @1:  74.81% | Train Acc @5:  99.02%
        Valid Loss: 0.609 | Valid Acc @1:  79.88% | Valid Acc @5:  99.64%
Epoch: 02 | Epoch Time: 29m 21s
        Train Loss: 0.628 | Train Acc @1:  78.89% | Train Acc @5:  99.64%
        Valid Loss: 0.622 | Valid Acc @1:  79.60% | Valid Acc @5:  99.54%
Epoch: 03 | Epoch Time: 30m 36s
        Train Loss: 0.623 | Train Acc @1:  79.14% | Train Acc @5:  99.67%
        Valid Loss: 0.617 | Valid Acc @1:  79.71% | Valid Acc @5:  99.64%
Epoch: 04 | Epoch Time: 30m 16s
        Train Loss: 0.588 | Train Acc @1:  79.98% | Train Acc @5:  99.66%
        Valid Loss: 0.604 | Valid Acc @1:  80.07% | Valid Acc @5:  99.56%
Epoch: 05 | Epoch Time: 29m 30s
        Train Loss: 0.565 | Train Acc @1:  80.68% | Train Acc @5:  99.77%
        Valid Loss: 0.571 | Valid Acc @1:  81.05% | Valid Acc @5:  99.68%
Epoch: 06 | Epoch Time: 29m 24s
        Train Loss: 0.538 | Train Acc @1:  81.75% | Train Acc @5:  99.79%
        Valid Loss: 0.546 | Valid Acc @1:  81.46% | Valid Acc @5:  99.65%
Epoch: 07 | Epoch Time: 29m 36s
        Train Loss: 0.509 | Train Acc @1:  82.34% | Train Acc @5:  99.85%
        Valid Loss: 0.539 | Valid Acc @1:  81.80% | Valid Acc @5:  99.80%
Epoch: 08 | Epoch Time: 29m 28s
        Train Loss: 0.470 | Train Acc @1:  83.62% | Train Acc @5:  99.90%
        Valid Loss: 0.543 | Valid Acc @1:  81.82% | Valid Acc @5:  99.78%
Epoch: 09 | Epoch Time: 29m 14s
        Train Loss: 0.427 | Train Acc @1:  84.72% | Train Acc @5:  99.92%
        Valid Loss: 0.549 | Valid Acc @1:  82.09% | Valid Acc @5:  99.74%
Epoch: 10 | Epoch Time: 30m 18s
        Train Loss: 0.398 | Train Acc @1:  85.63% | Train Acc @5:  99.95%
        Valid Loss: 0.555 | Valid Acc @1:  82.10% | Valid Acc @5:  99.76%
```

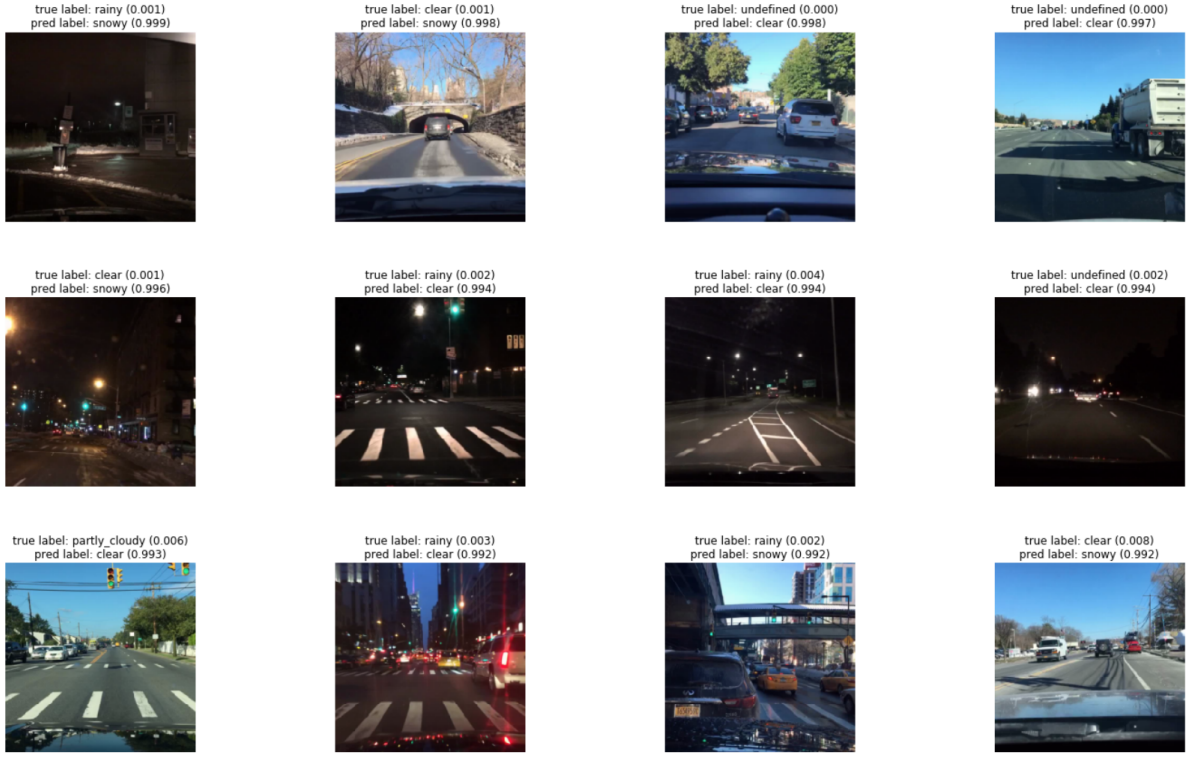
The test accuracies after training are slightly lower than the validation accuracies, but not by a significant margin. Test Loss: 0.531 | Test Acc @1: 82.34% | Test Acc @5: 99.85%

Ⅴ. Performance

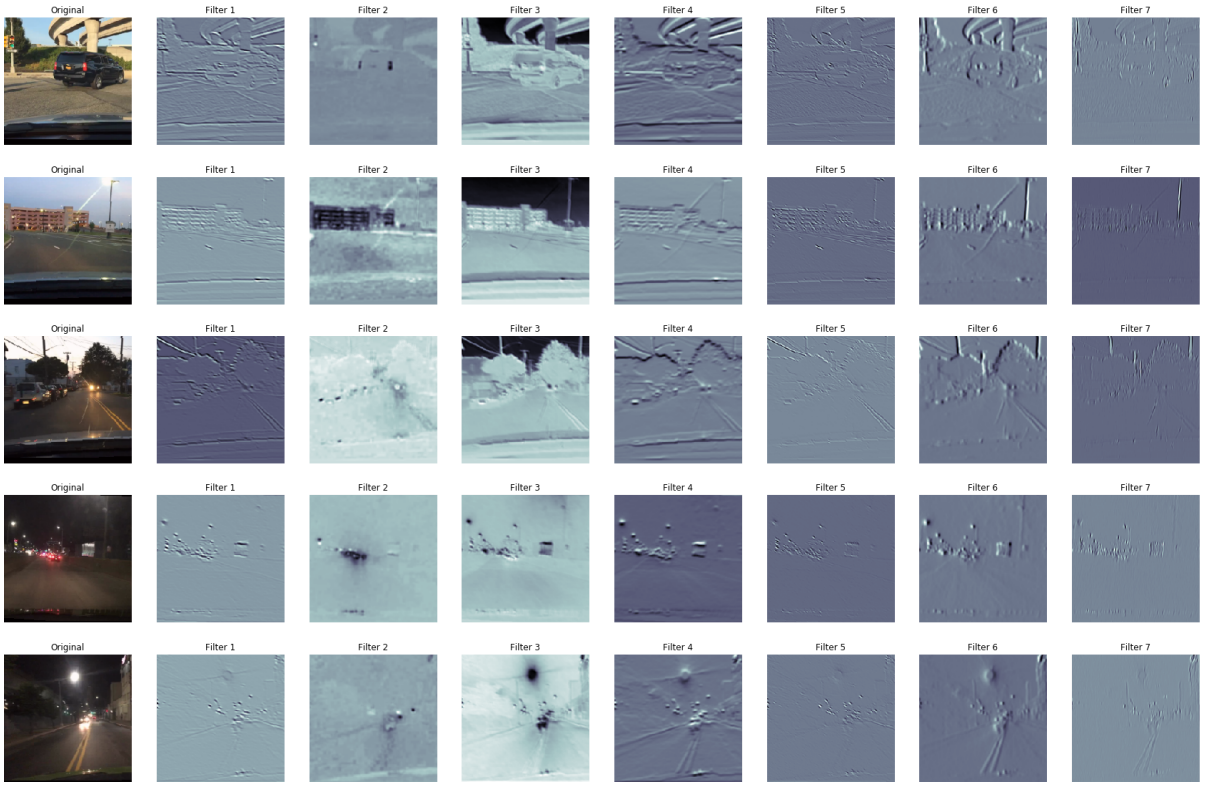
To retrieve the predicted labels, we first need to get the predictions for each image in the test set. This is then used to create a confusion matrix.



We can then gather all of the correct predictions, filter them out, and rank all of the incorrect predictions according to their level of confidence in their incorrect prediction. The most erroneously predicted images, as well as the anticipated and actual classes, can then be plotted.



After that, we plot the results of a couple of the images after they've gone through the first convolutional layer. The filters perform a wide range of image processing tasks, from edge recognition to color inversion, as can be seen.



VI. Conclusion

On a dataset with seven classes, we fine-tuned a pre-trained model to achieve 82 percent top-1 accuracy and 95 percent top-5 accuracy, as demonstrated in the above results. As a result, our model functioned admirably.

On the t-SNE plot, the classes are clearly separated, which is usually a good sign.

We can observe that the inaccurate predictions are usually sensible, e.g. undefined and clear, rainy and clear, from the names of the classes (and a little picture searching).

The image with the highest erroneous predictions could alternatively be a mislabeled example with unclear labels.

VII. Reference

1. He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778. 2016.
2. Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).
3. Link for reproducible files(code): https://drive.google.com/file/d/1mY915CjlYnQQ6AOouO3VyOxQ-ugXsiNl/view?usp=sharing