Division of Engineering Science
UNIVERSITY OF TORONTO

ROB501: Computer Vision for Robotics

Project #5: Deep Visual Sun Sensing

Fall 2019

Overview

Deep learning (i.e., deep neural networks) has exploded in popularity recently, in part due to the impressive performance of deep neural nets when applied to many challenging robotics problems. In this project, you will use a specific type of deep neural network (a convolutional neural net or CNN) to predict the direction of the sun from a single RGB image. Knowing the direction of a landmark like the sun can be helpful for robot localization (to reduce drift). The goals are to:

- introduce several basic concepts related to the design of CNNs, and
- provide some practical experience working with such a network using the PyTorch framework.

The due date for project submission is **Wednesday, December 4, 2019, by 11:59 p.m. EDT**. All submissions will be in Python 3 via Autolab (more details will be provided in class and on Quercus); you may submit up to five times prior to the deadline.

Details

While you could try to train a CNN to regress a 3D unit vector, this is a bit complicated. Instead, you will train a small network to predict only the azimuth angle θ_{az} of the sun (in degrees) by classifying images into discretized bins of azimuth angles (e.g., 0-45 deg, 45-90 deg, etc.).

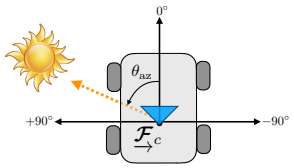


Figure 1: In this assignment you will train a CNN to estimate the azimuth angle of the sun relative to the camera.

For this assignment you will use real image data from the well known KITTI dataset and the PyTorch machine learning framework to implement a CNN in Python. Both are discussed in the following sections.

The KITTI Dataset

The Karlsruhe Institute of Technology and Toyota Technical Institute at Chicago (KITTI) dataset is a widely used dataset in autonomous driving research. It includes RGB and grayscale images, inertial measurements and lidar point cloud data collected from a moving vehicle in Karlsruhe, Germany in 2011. In this assignment you will be training and testing a model using a 4541-image sequence from the KITTI dataset, of which 80% will be used to train the model, approximately 16% will be used as validation data, and approximately 4% will be held out as test data for grading purposes.

PyTorch

PyTorch is a popular open source Python-based machine learning framework. It provides a wide range of tools for learning, and can be used to implement and train CNNs for computer vision applications. For this project, you will create a simple feed-forward network by specifying a sequence of neural network layers.

Begin by installing PyTorch from here. While having a GPU significantly reduces training time, it is not necessary for this assignment (all processing can be done on your CPU). A brief overview of the implementation and training a neural network using PyTorch is available here. Full documentation for the different types of layers available can be found at: `https://pytorch.org/docs/stable/index.html`.

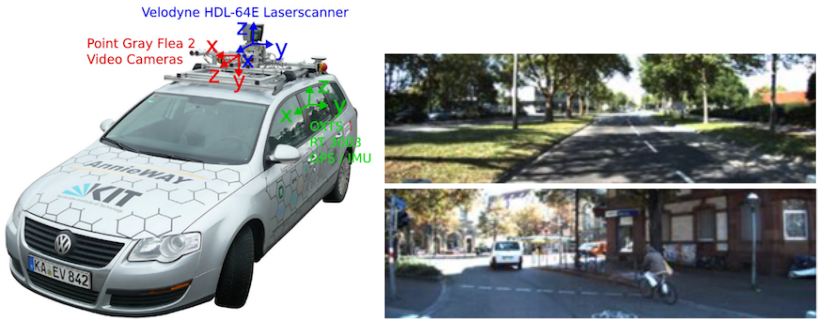
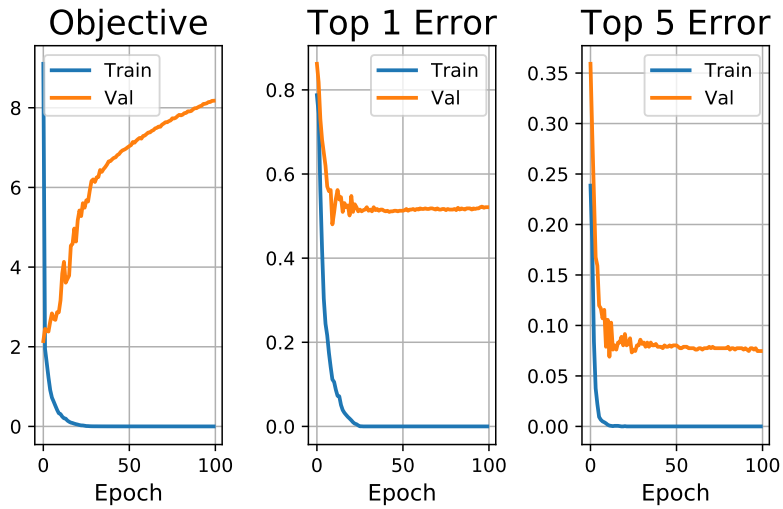


Figure 2: The KITTI dataset. (Left) The sensor suite used to collect the data. (Right) Sample images.

Getting Started

A working example is already provided in `sun_cnn.py` to get you started classifying images into 45-degree bins of sun azimuth angle (8 different classes). There are a number of important sections in the code, all of which are documented. Read through the comments to get a feel for what the different parts do, then try running `sun_cnn.py`. After a few minutes of training, you should see something like this (these plots are saved as `net-train.pdf` in the project folder):



The "Objective" plot in the left pane shows the value of the loss function after each training epoch, or pass through the training data. The "Top 1 Error" plot in the middle pane tells us how often the highest scoring guess is wrong, while the "Top 5 Error" plot in the right pane tells us how often the five highest scoring guesses were all wrong. All of these values are shown for the training set (blue) and the held-out validation set (orange) for each epoch of training (full pass through the training set).

For our purposes, we're interested in the top 1 error, and particularly the top 1 error on the validation set, since this tells us something about how well the learned weights might generalize to new image data. The top 5 error is not important in this case, but can be a useful diagnostic tool to see if your network is doing something (un)reasonable.

Network structure

We can see that the basic network achieves at best a paltry 52% validation accuracy (48% top 1 error on the validation set) around epoch 10. We can do better, but first let's take a look at the network structure to see what it's doing.

```
class CNN(torch.nn.Module):
  def __init__(self, num_bins):
    super(CNN, self).__init__()

    ### Initialize the various Network Layers
    self.conv1 = torch.nn.Conv2d(3, 16, stride=4, kernel_size=(9,9))
    self.pool1 = torch.nn.MaxPool2d((3,3),stride=3)
    self.relu = torch.nn.ReLU()
    self.conv2 = torch.nn.Conv2d(16, num_bins, kernel_size=(5,18))

  ### Define what the forward pass through the network is
  def forward(self, x):
    x = self.conv1(x)
    x = self.pool1(x)
    x = self.relu(x)
    x = self.conv2(x)
    x = x.squeeze() #(Batch_size x num_bins x 1 x 1) to (Batch_size x num_bins)
    return x
```

This simple baseline network has four layers (see Figure 3) – a convolutional layer, followed by a max pooling layer, followed by a rectified linear (relu) layer, followed by another convolutional layer. This last convolutional layer might be called a "fully connected" or "FC" layer because its output has a spatial resolution of 1×1, and it's a function of all the outputs of the previous layer.

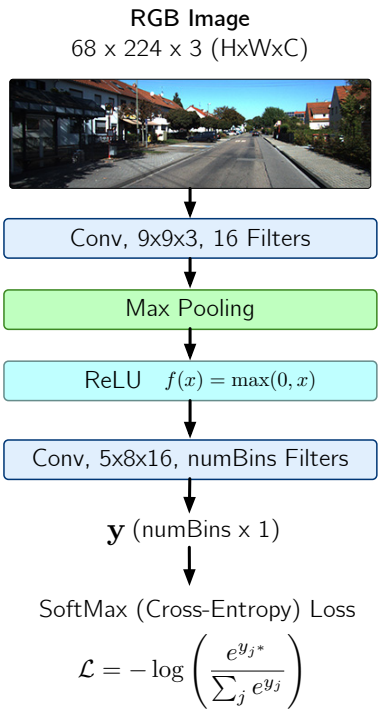


Figure 3: The initial network structure.

Convolutional Layer

Let's look at the first convolutional layer. The "weights" are the filters being learned. The first layer has a 9×9 spatial resolution, a filter depth of 3 (because the input images have 3 channels, R, G, B), and is composed of 16 filters. The network also learns a bias or constant offset to associate with the output of each filter.

Max Pooling Layer

The next layer is a max pooling layer. It will take a max over a 3×3 sliding window and then subsample the resulting image/map with a stride of 3. Thus the max pooling layer will decrease the spatial resolution by a factor of 3 according to the stride parameter. The filter depth will remain the same (16). There are other pooling possibilities (e.g., average pooling) but we will only use max pooling in this assignment.

Non-Linearity

The next layer is the non-linearity. Any values in the feature map from the max pooling layer which are negative will be set to 0. There are other non-linearity possibilities (e.g., sigmoid) but we will use only rectified linear layers in this assignment.

Note that the pool layer and relu layer have no learned parameters associated with them. We are hand-specifying their behaviour entirely, so there are no weights to initialize as in the convolutional layers.

Fully-Connected Layer

Finally, we have the last layer which is convolutional (but might be called "fully connected" because it happens to reduce the spatial resolution to 1×1). The filters learned at this layer operate on the rectified, subsampled, maxpooled filter responses from the first layer. The output of this layer must be 1×1 spatial resolution (or "data size") and it must have a filter depth of `numBins` (corresponding to the number of bins of azimuth angles). Here, 5×18 is the spatial resolution of the filters, 16 is the number of filter dimensions that each of these filters take as input and `numBins` is the number of output dimensions. The number of dimensions coming out of a layer must match the number of input dimensions of the next layer.

Loss Function

At the end of our network we add one more layer which is only used for training. This is the "loss" layer. There are many possible loss functions but we will use the "softmax" loss (a.k.a. "cross-entropy" loss) for this project:

$$\mathcal{L} = -\log\left(\frac{e^{y_{j^*}}}{\sum_j e^{y_j}}\right) \quad \text{or equivalently} \quad \mathcal{L} = -y_{j^*} + \log\sum_j e^{y_j}$$

where y_j is the j-th element of the vector of class scores $\mathbf{y}$ output by the network and y_{j^*} is the class score assigned to the correct class j^* (based on the ground truth labels).

This loss function will measure how badly the network is doing for any input (i.e., how different its final layer activations are from the ground truth, where ground truth in our case is which sun azimuth bin each image corresponds to). The network weights will update, through back propagation, based on the derivative of the loss function. With each training batch the network weights will take a tiny gradient descent step in the direction that should decrease the loss function (but isn't actually guaranteed to, because the steps are of some finite length).

Visualizing the Network

How did we know to make the final layer filters have a spatial resolution of 5×18? It's not obvious because we don't directly specify output resolution. Instead it is derived from the input image resolution and the filter widths, padding, and strides of the previous layers. Luckily we can visualize the output size at each layer to help us figure this out. This is what the output looks like—it will show up any time you run `sun_cnn.py`, if you uncomment lines 62-64 (note that `torchsummary` must be installed for this to work):

For each layer this visualization shows several useful attributes. "Layer (type)" shows the sequence of layers that our network is composed of. "Output Shape" is the spatial resolution of the feature maps at each level. Notice that `layer4` (the final convolution layer) has an "output shape" of [-1, 8, 1, 1]. This corresponds to a 1×1 map whose 8 channels are the "scores" for each of the 45-degree azimuth bins (the -1 corresponds to the batch size, which can be specified within the dataloader). The data depth can change if you change the size and number of bins, but the data size must always be 1 for the final layer. In this network and most deep networks this will decrease as you move up the network. Finally this visualization shows us that the network has 15,000 trainable parameters, the vast majority of them associated with the last convolutional layer.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 16, 15, 54]           3,904
         MaxPool2d-2           [-1, 16, 5, 18]               0
              ReLU-3           [-1, 16, 5, 18]               0
            Conv2d-4             [-1, 8, 1, 1]          11,528
================================================================
Total params: 15,432
Trainable params: 15,432
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.17
Forward/backward pass size (MB): 0.12
Params size (MB): 0.06
Estimated Total Size (MB): 0.35
----------------------------------------------------------------
```

Performance Improvements

There are a number of possible ways to improve the performance of the baseline network. For example, you can do one or more of the following:

- *Make the network deeper.* CNNs excel at exploiting hierarchical information in images. Try adding another convolutional layer to the network. In fact, you probably don't want to add just a convolutional layer, but also another max pooling layer and a relu layer as well. For example, you might add a convolution layer with a 5×5 kernel followed by a max-pool over a 3×3 window with a stride of 1. You can adjust the padding and strides of each layer and the spatial resolution of the final convolutional layer – just make sure the final convolutional (fully connected) layer shows a data size of 1.
- *Use batch normalization.* Deeper networks are harder to train and more sensitive to initialization. Batch normalization can help by making it easier for gradient information to flow through the network. This may improve accuracy, but it will also allow you to use much higher learning rates. Try adding batch normalization layers after each convolutional layer except the last one (the "fully connected" layer). See the PyTorch documentation for how to implement a batch normalization layer to the network.
- *Zero-center the input data.* Classifiers often work better when the input data is zero-mean. You can do this by computing and subtracting the mean of the training set from both the input and validation images during the data loading/pre-processing stage.
- *Regularize the network using dropout.* If you train the network for enough epochs, you will notice that the training error continues to decrease even though the validation error stays flat. This is because the network is *overfitting* to the training data (i.e., the network has learned weights which can perfectly recognize the training data, but those weights don't generalize to held out test data). The best way to resolve this is to add more training data, but since we don't have that, we will use *dropout* regularization instead. See the PyTorch documentation for how to implement a dropout layer to the network.

  Dropout randomly turns off network connections at training time to fight overfitting. This prevents a unit in one layer from relying too strongly on a single unit in the previous layer. Dropout regularization can be interpreted as simultaneously training many "thinned" versions of your network. At test time, all connections are restored which is analogous to taking an average prediction over all of the "thinned" networks. The dropout layer has only one free parameter – the dropout rate – the proportion of connections that are randomly deleted. Experiment with p={0.3, 0.4, 0.5}.

- *Adjust the learning rate.* Deep networks can be sensitive to the learning rate. If things aren't working, try adjusting the learning rate up or down by factors of 10. If the objective remains exactly constant over the first dozen epochs, the learning rate might have been too high and "broken" some aspect of the network. If the objective spikes or even becomes `NaN` then the learning rate may also be too large. However, a very small learning rate requires many training epochs.
- *Train for more epochs.* If the validation objective is still decreasing when training stops, you may want to train for more epochs. The appropriate number of training epochs will depend on your chosen learning rate, so some experimentation is required.

Part 1: Improving Network Performance

With all of the above said, it's time to build a better network! Modify the network in `sun_cnn.py` to improve the validation set accuracy—name this new network file `sun_cnn_45.py`. The Python script will automatically save the model that results in the highest validation set accuracy to `best_model_45.pth`.

When you've trained your model, run `test_sun_cnn.py` to reload your model and evaluate it on the held-out test set. The `test_sun_cnn.py` script outputs a text file called `predictions_45.txt`; you should submit the following for grading:

- The modified network file `sun_cnn_45.py` and the plain text file `predictions_45.txt` that contains a list of the 200 test set predictions. Autolab will return the accuracy: it should be at least 70% (i.e., less than 30% top 1 error), using 45-degree bins for the sun azimuth.

To reach this performance goal, you should choose and implement two or three of the performance improvements listed above (hint: start with pre-processing). You should be able to get good results within 30 epochs and under 10 minutes of training (depending on your CPU).

Part 2: Including More Azimuth Bins

The 45-degree azimuth discretization is fairly coarse, and it's possible to do better. For this portion of the project, you should modify the network (just change line 62 in `sun_cnn.py`) to use 20-degree azimuth bins instead (18 bins in total instead of 8) and to improve its performance——name this new network file `sun_cnn_20.py`

When you've trained your model, run `test_sun_cnn.py` to reload your model and evaluate it on the held-out test set. Remember to update the bin size to 20 in this script (change line 10)! Then `test_sun_cnn.py` will output a text file called `predictions_20.txt`; you should submit the following for grading:

- The modified network file `sun_cnn_20.py` and the plain text file `predictions_20.txt` that contains a list of the 200 test set predictions. Autolab will return the accuracy: it should be at least 60% (i.e., less than 40% top 1 error), using 20-degree bins for the sun azimuth.

Again, to reach this performance goal, you should choose and implement two or three of the performance improvements listed above. Training may also take longer in this case.

Grading

Points for each portion of the project will be assigned as follows:

- Improved Sun-CNN network – **25 points** (up to 5 tests, 25 points)

  The trained network must achieve at least 70% test accuracy (i.e., less than 30% top 1 error on the test set), using 45 degree bins for the sun azimuth.

- Thin-Binned Sun-CNN network – **25 points** (up to 5 tests, 25 points)

  The trained network must achieve at least 60% test accuracy (i.e., less than 40% top 1 error on the test set), using 20 degree bins for the sun azimuth.

Total: **50 points**

Grading criteria include: correctness and succinctness of the implementation of support functions, proper overall program operation, and code commenting. Please note that we will test your code *and it must run successfully*. Code that is not properly commented or that looks like 'spaghetti' may result in an overall deduction of up to 10%.