

**PROJECT REPORT
ON
SENSOR DATA COLLECTION AND DEEP LEARNING
TECHNIQUES FOR IMAGE CLASSIFICATION**

**Submitted By
Saksham Gupta
2th Year
Computer Science
Chandigarh College of Engineering and Technology (Degree Wing)**

**Under the guidance of
Mr. Ritesh Kumar**



**Department of Food and Agrionics, Technology Block
CSIO-CSIR
Chandigarh
July 2018**

Acknowledgement

I would like to express my deep and sincere gratitude to my projet instructors, Mr. Ritesh Kumar who gave me the opportunity to work on this project on the topic 'DEEP LEARNING TECHNIQUES FOR IMAGE CLASSIFICATION AND SENSOR DATA COLLECTION' which strengthened my knowledge and understanding of this topic and the entire field in general.

Further, it allowed me to go through a lot of research as a result of which we learnt many new things. I am also duty to record our thankfulness to people who have previously contributed to this project. Their work gave me better insight to the path we followed for completion of this project.

Also, I am grateful to the technical team at CSIO-CSIR, for providing an efficient working system and environment. Lastly I would thank my parents and friends for their kind co-operation and encouragement which helped me in completion of this project.

With warm regards,
Saksham Gupta

Abstract

There are three applications I covered during the tenure of my training, two in sensor data collection and one in Image Classification.

The first software acts as an interface between the Taste Sensor device and data collection along with data plotting. The software provides a GUI where the user can input the metadata including date, time, concentration, name and type of solution. The software collects data from the sensor and stores the data in a CSV file along with the metadata provided by the user.

The second software helps in the plotting and representation of data collection in asc files from the UV wavelength detection sensor. The software provides means to plot multiple sample values on a single plot to compare them on a unified scale.

In image classification the data was acquired from the files and was plotted on graphs of different classifications including raw data, normalized data and the ratio of data.

The image classification involved a dataset of road faults of over 10,000 images separated by damages. These images were passed through various CNN layers and max pooling layers in order to classify the images.

Sensor Data Collection

Application I – Taste Sensor Live Data Streaming

Usage:

This software acts as an interface between the Taste Sensor device and data collection along with data plotting. The software provides a GUI where the user can input the metadata including date, time, concentration, name and type of solution. The software collects data from the sensor and stores the data in a CSV file along with the metadata provided by the user.

In addition to data collection and storage, the software also provides a live streaming of the sensor data on a graph which can be controlled by start/stop actions and calibrated using a times in the GUI. The data plots a maximum of fifty values at a given time, with each new value of the seven sensors replacing the previous values.

The graph plots the data from seven sensors on a single plot to ensure comparison and each sensor data is stored in separate column in the CSV file.

Input/Output:

Input required from the user:

1. Port number of the connected sensor to be added in the python script. (fetch_th.py)
2. Date, time, concentration, name and type of solution to be added in the GUI.
3. All the seven sensors of the sensor connected in appropriate manner.

Output provided to the user:

1. CSV file containing the metadata input by the user along with the sensor data in separate columns (S0 – S6). The file would be named after the current date and time.
2. A single plot providing the live data stream collected from the sensor along with the appropriate labels and legend.

Language and Libraries:

Python: Data fetch and GUI script, along with plotting code and Sensor access

Python Libraries: OS, Matplotlib, csv, serial, PyQt5, datetime, time, numpy, threading

Arduino Language (C/C++ functions): To gather the sensor data from the arduino device

Functions/Classes:

Events_th.py:-

Class: Ui_Form

1. **generateMeta():** Gather the current date and time, metadata (including user inputs stated above) from UI forms and store them in class variables.
2. **writeMeta():** Create a new CSV file which is named according to the current system time and write the metadata collected in class variables to the file. Also, write the sensor heading in the file (S0-S6).
3. **writeSensorMeta():** Call functions generateMeta() and writeMeta().
4. **graph():** Create a new row in the csv file and add the latest sensor data received to the file. Provide the title, grid, x and y labels to the plot. Add a condition to display a maximum of sensor values from each sensor at a single time. Create plots for the seven sensor values.
5. **Time():** Create an hour, minute and second time using the pyqt backward timer and display the timer on the GUI LCD.
6. **Reset():** Set the values of the time to 0 and clear the GUI. Generate a new filename for the csv file to create a new file during data collection.
7. **timerStart():** start the pyqt timer with 1 second.
8. **stopGraph():** Call the flag function in fetch module to stop the execution of data collection thread. Add a delay to ensure that thread gets properly terminated. Close the port to the sensor. Clear the flag so that data can be collected again. Enable the reset button on the GUI.
9. **setupUi():** Create and position the GUI containing the Labels, Input fields, Timer, LCD screen and Start, Stop and Reset buttons.
10. **retranslateUi():** Set the label values for the GUI and provide values for the dropdown menu.

fetch_th.py

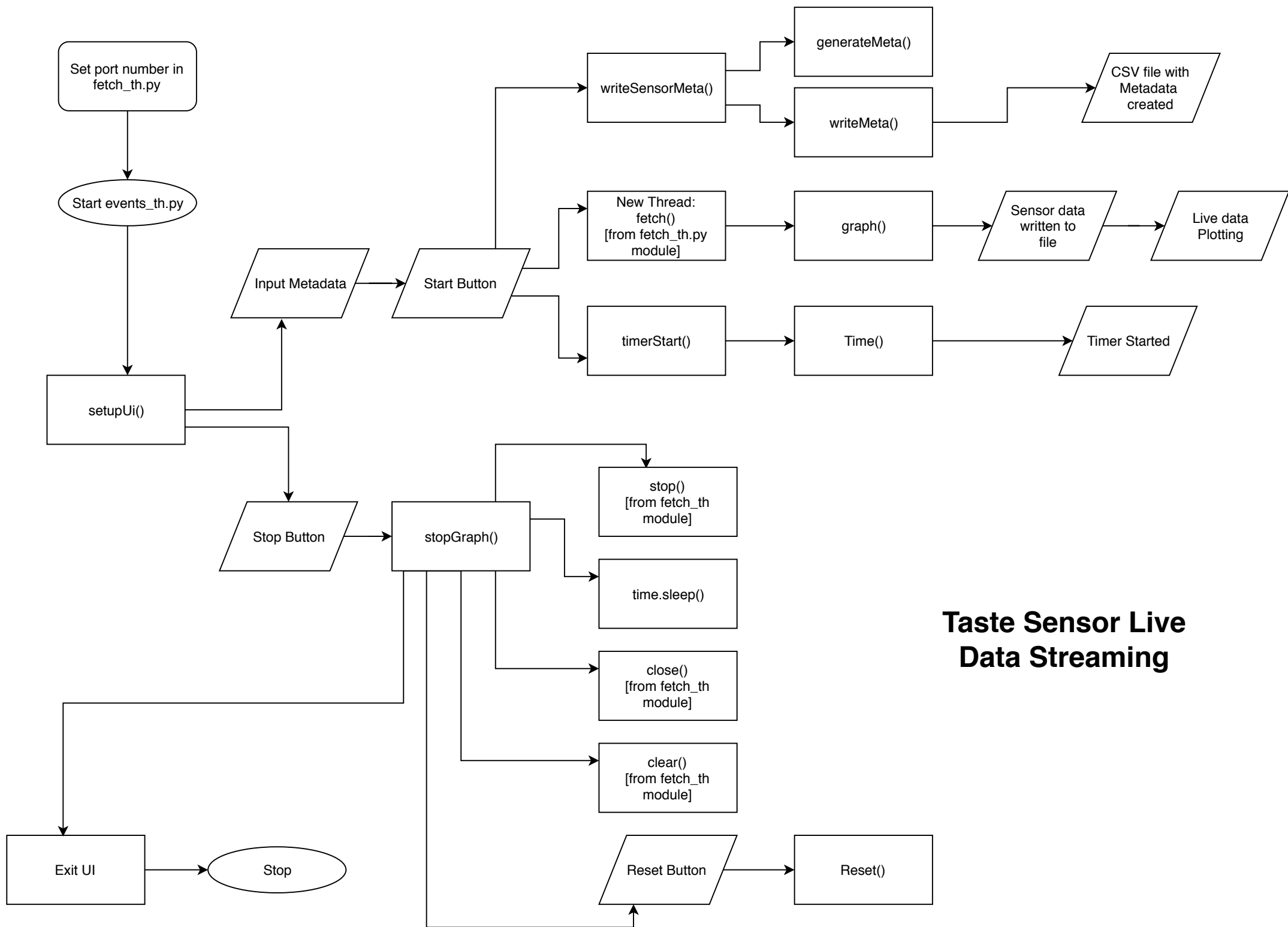
Class: SerialData

1. **stop():** Set flag to stop the thread which is fetching the data from the sensor.
2. **clear():** Clear the flag to start running the script again
3. **fetch():** Open a port to the arduino and get the sensor data string. Format and clean the data string and split it's values into seven sensor values. Further signal that the sensor data has changed and call the graph function.

Working:

1. First, we connect the arduino taste sensor to the USB port of the system.
2. We open the arduino software and note the port number of the connected sensor. Then we run the arduino program to start fetching the data from the sensor.
3. Next, we open the fetch_th.py file and change the port number to the one noted in the above step.
4. We can then run the events_th.py script using any python interpreter or console.
5. When the script is run, the User interface GUI is visible, which consists of the different user inputs like name, date, time, solution type and concentration. The GUI also shows a timer currently set to 0.
6. Once the input fields are filled, the user can click start to start fetching, storing and plotting the sensor data.
7. Once start button is pressed, the fetch function of the fetch_th.py module is called which starts receiving the data from the sensor. Along with this, the timer also starts increasing the time by 1 second. The fetching operation occurs on a separate thread and is independent from the GUI function.
8. Whenever the data value of the fetch function changes, the graph function is called which plots the latest data on the pre-existing plot with a maximum of 50 values plotted at one time.
9. The write meta functions meanwhile write the user inputs to a new file and start storing the sensor data.

10. When the user presses the stop button a flag is set to stop the execution of the fetch thread and after a delay, the port to the sensor is also closed. This ensures that every time a new file is formed whenever some new data is to be taken.
11. After pressing the stop button, the reset button becomes active which can be used to clear the input boxes and restart the process.



Taste Sensor Live Data Streaming

Application II – UV Data Plotting and Representation

Usage:

This software helps in the plotting and representation of data collection in asc files from the UV wavelength detection sensor. The software provides means to plot multiple sample values on a single plot to compare them on a unified scale.

The data is plotted in a raw, normalized and ratio format and each plot displays all the sample values given as input in the data files. Along with data plotting, the software also provides the areas under the curve for all the samples.

1. RAW Data: Direct Data Values from the data files
2. Normalized Data: $(Y - Y_{\min}) / (Y_{\max} - Y_{\min})$, for all values of Y over x
3. Ratio Data: Y / Y_{\max} , for all values of Y over x

Input/Output:

Input: Data Files with the raw sensor data from UV sensor

Output: Graph containing plots of raw data, normalized data and ratio data.

Language and Libraries:

Python: Scripting, Plotting and Representation

Python Libraries: scipy.integrate, Matplotlib, CSV

Functions/Classes:

Class: fileplot

1. **plot():** Wrapper function to call functions to plot raw, normalized and ratio data.
2. **generateLabel():** Create label names for the legend by splitting and formatting the input filenames.
3. **readFile():** Open the input file and read every data row and store the values of X and Y in local variable lists.
4. **generateNormalized():** Create the normalized values of Y using the formula mentioned above.

5. **generateRatio()**: Create ratio data using the formula mentioned above.
6. **area()**: Calculate the area of the plot area under the curve for a particular sample.
7. **plotRaw()**: Generate the plot for displaying raw sensor data.
8. **plotNormalized()**: Generate the plot for displaying normalized sensor data.
9. **plotRatio()**: Generate the plot for displaying ratio sensor data.
10. **showPlot()**: Show plot window on the console.

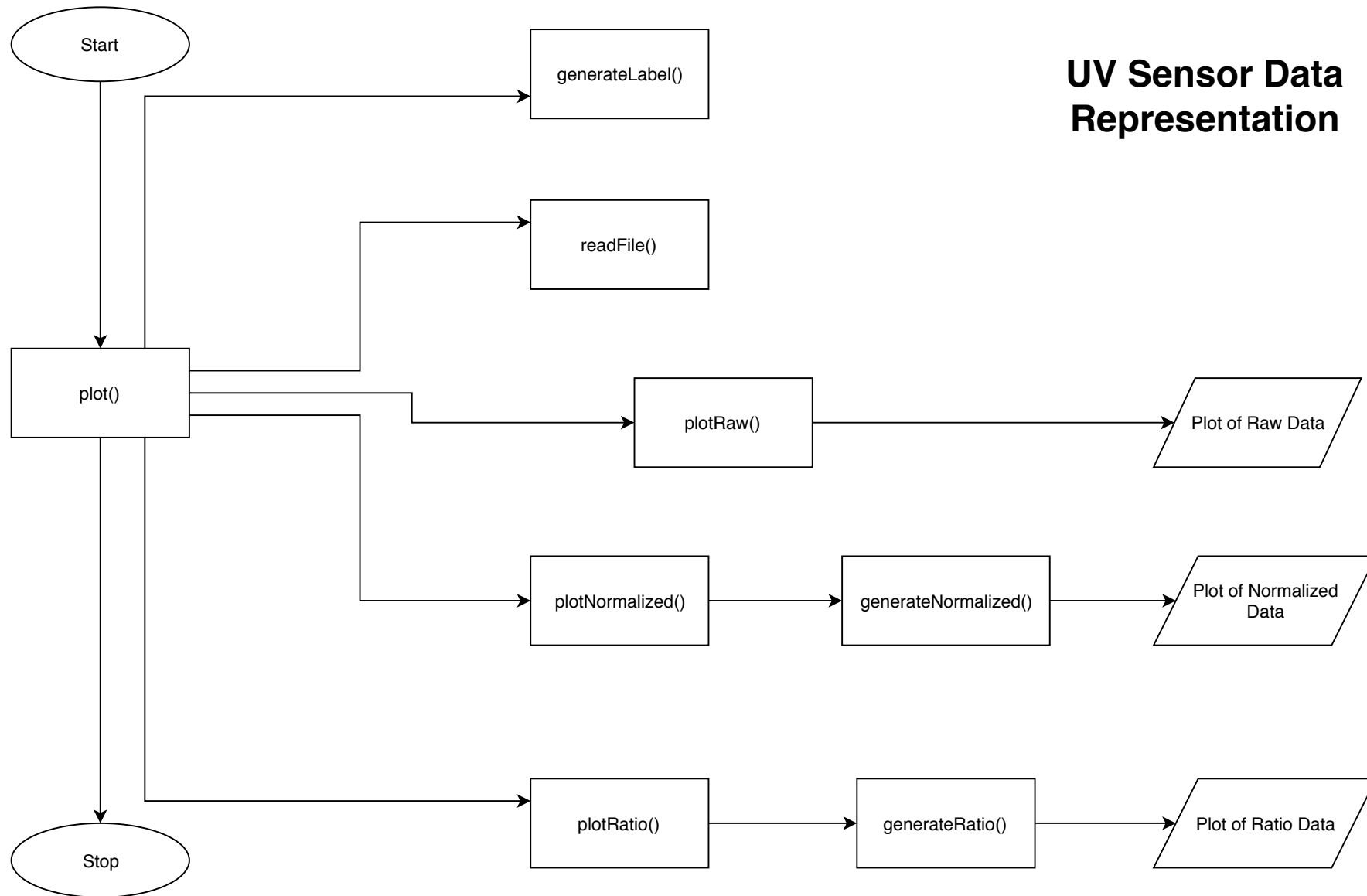
Working:

Initially we keep all the asc data files in the same folder as the python script.

Further, we use the plot function of the fileplot class to plot the graphs for the various samples. We can do this by calling the plot function as many times as the number of sample files that are required to be plotted.

We can then save the script and run it, which will return 3 plot windows, displaying Raw, Normalized and Ratio of data respectively.

UV Sensor Data Representation



Deep Learning Image Classification

Application I – Road Damage/City Classification

Theory:

Convolutional Neural Networks (CNN):

This word is composed of 2 main words : convolution and neural network. Let's define them separately and then see how these 2 things actually connect. Let's first see what does a neural network mean.

Convolution:

For computer an image is nothing but a matrix of pixel values. So computer understands images something like this, for any image it is just a matrix of zeros and ones . So, machine has to interpret as much as it can by looking at this matrix only. So it is not as easy task as it seemed earlier. There is whole another field of image processing which only specializes in extracting more and more information from an image. People use various filters to get an idea of sharp edges present in that image.

A convolutional neural network consists of several layers, there are 3 main types of layers:

1. Convolutional Layer:

A filter of given size convolves over the input and produces a matrix. Suppose the layer takes in a volume of $w * H * D$, and hence it needs 4 hyperparameters:

K: Number of filters

F: The spatial extent

S: Stride

P: Amount of zero padding

2. Pooling Layer:

Generally the image has very large dimensions with respect to its features and make it very costly in terms of computation. Thus the pooling layer helps to reduce these dimensions maintaining the features of the image. Reducing the parameters in pooling will help avoid overfitting. It takes an input of a volume of size $A \times B \times C$ and needs 2 hyperparameters: F the spatial extent and S the stride. It outputs a volume of size $D \times E \times F$ where: $D = (A - F) / S + 1$, $E = (B - F) / S + 1$ and $F = C$

3. Fully Connected Layer:

The layer of the network is known to be fully connected layer if the units have all the interconnections to all the activations existing in the previous layer. Note that this layer is also called a output layer. Now we want a class at the end of classification. Intermediate pooling and convolution layers are useful for extracting features and hence result in reduction of parameters of the original image. So in order to classify

in say n classes, we make use of fully connected layer of size n . Output layer possesses a loss function. For example, categorical cross entropy loss function. When single forward pass completes, model goes to backpropagation which was already discussed in great detail for updating biases and weights and calculating errors and losses and effectively decreasing them in each iteration

Learning Rate:

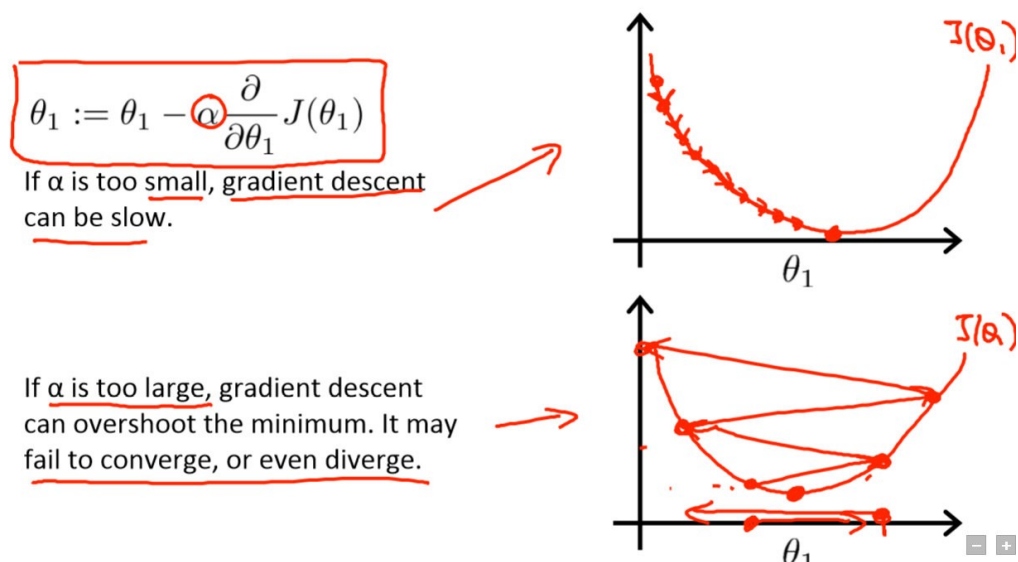
The learning rate is one of the most important hyper-parameters to tune for training deep neural networks.

How does learning rate impact training?

Deep learning models are typically trained by a stochastic gradient descent optimizer. There are many variations of stochastic gradient descent: Adam, RMSProp, Adagrad, etc. All of them let you set the learning rate. This parameter tells the optimizer how far to move the weights in the direction of the gradient for a mini-batch.

If the learning rate is low, then training is more reliable, but optimization will take a lot of time because steps towards the minimum of the loss function are tiny.

If the learning rate is high, then training may not converge or even diverge. Weight changes can be so big that the optimizer overshoots the minimum and makes the loss worse.



Gradient descent with small (top) and large (bottom) learning rates. Source: Andrew Ng's Machine Learning course on Coursera

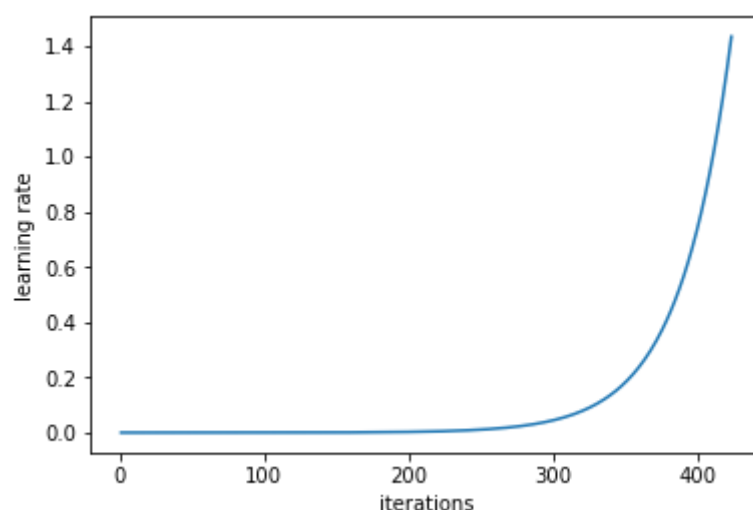
The training should start from a relatively large learning rate because, in the beginning, random weights are far from optimal, and then the learning rate can decrease during training to allow more fine-grained weight updates.

There are multiple ways to select a good starting point for the learning rate. A naive approach is to try a few different values and see which one gives you the best loss without sacrificing speed of training. We might start with a large value like 0.1, then try exponentially lower values: 0.01, 0.001, etc. When we start training with a large learning rate, the loss doesn't improve and probably even grows while we run the first few iterations of training. When training with a smaller learning rate, at some point the value of the loss function starts decreasing in the first few iterations. This learning rate is the maximum we can use, any higher value doesn't let the training converge. Even this value is too high: it won't be good enough to train for multiple epochs because over time the network will require more fine-grained weight updates. Therefore, a reasonable learning rate to start training from will be probably 1–2 orders of magnitude lower.

A better and More Efficient Way

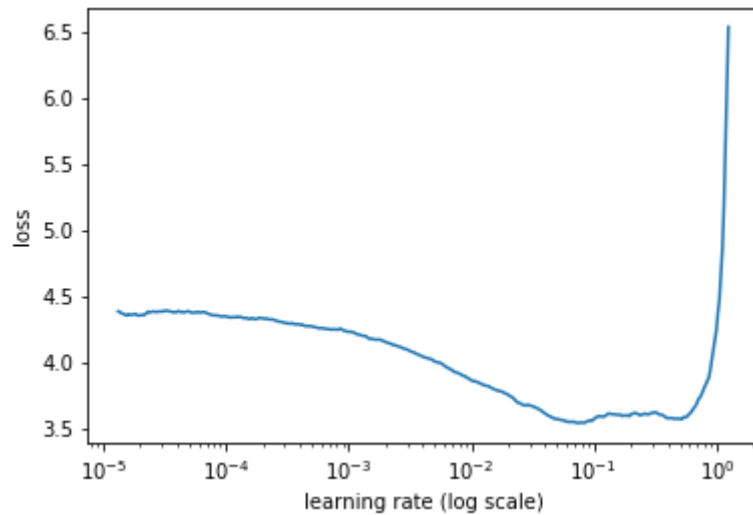
Leslie N. Smith describes a powerful technique to select a range of learning rates for a neural network in section 3.3 of the 2015 paper "[Cyclical Learning Rates for Training Neural Networks](#)".

The trick is to train a network starting from a low learning rate and increase the learning rate exponentially for every batch.



Learning rate increases after each mini-batch

Record the learning rate and training loss for every batch. Then, plot the loss and the learning rate. Typically, it looks like this:

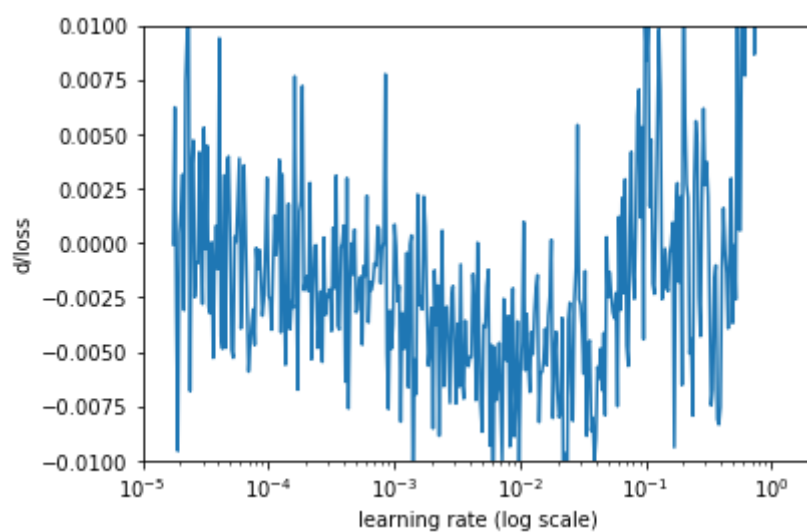


The loss decreases in the beginning, then the training process starts diverging

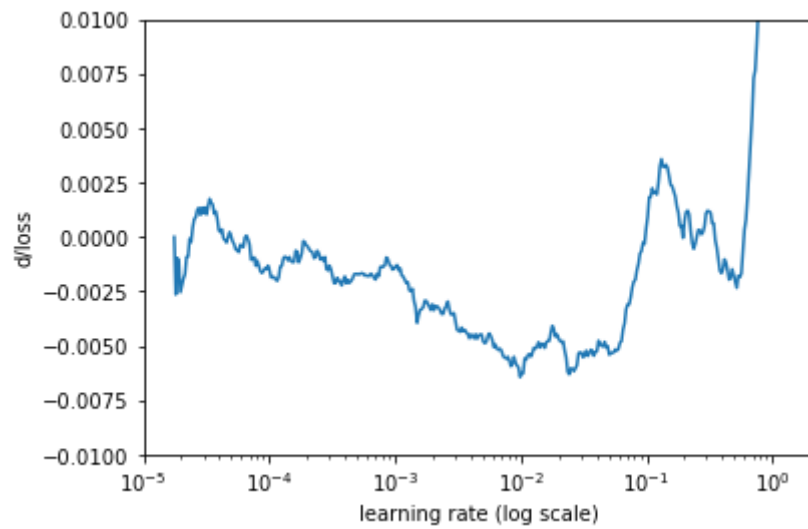
First, with low learning rates, the loss improves slowly, then training accelerates until the learning rate becomes too large and loss goes up: the training process diverges.

We need to select a point on the graph with the fastest decrease in the loss. In this example, the loss function decreases fast when the learning rate is between 0.001 and 0.01.

Another way to look at these numbers is calculating the rate of change of the loss (a derivative of the loss function with respect to iteration number), then plot the change rate on the y-axis and the learning rate on the x-axis.



Rate of change of the loss



Rate of change of the loss, simple moving average

This concept has been readily made available by the fastai library developed by Prof. Jeremy Howard, which utilized this technique to give a plot of the learning rate and loss function.

	id	damage
0	train_Adachi_00001	D44
1	train_Adachi_00002	D01
2	train_Adachi_00003	D01
3	train_Adachi_00004	D01

damage	
D01	1793
D44	1788
D20	1327
D00	1018
D43	487
D10	299
D11	286
D40	233

```

In [30]: tfms = tfms_from_model(arch, sz, aug_tfms = transforms_side_on, max_zoom = 1.5)
data = ImageClassifierData.from_csv(PATH, 'train', f'{PATH}labels.csv', test_name = 'test',
                                val_idxs=val_idxs, suffix='.jpg', tfms = tfms, bs = bs)


In [31]: fn = PATH+data.trn_ds.fnames[0]
fn

Out[31]: 'data/road-damage/train/train_Adachi_00002.jpg'

In [32]: img = PIL.Image.open(fn);img

Out[32]:

```



damage	
D01	1793
D44	1788
D20	1327
D00	1018
D43	487
D10	299
D11	286
D40	233

Out[32]:





```
In [33]: size_d = {k: PIL.Image.open(PATH+k).size for k in data.trn_ds.fnames}

In [34]: row_sz, col_sz = list(zip(*size_d.values()))

In [35]: row_sz = np.array(row_sz); col_sz = np.array(col_sz)

In [36]: plt.hist(row_sz)

Out[36]: (array([ 0., 0., 0., 0., 0., 5785., 0., 0., 0., 0.]),
array([599.5, 599.6, 599.7, 599.8, 599.9, 600., 600.1, 600.2, 600.3, 600.4, 600.5]),
<a list of 10 Patch objects>)
```



Category	Number of people (millions)
Did not vote	~5800

```

tfms = tfms_from_image_size(img, sz, aug_tfms = transforms.Compose([
    data = ImageClassifierData.from_csv(PATH, 'train', f'{PATH}labels.csv', test_name = 'test', num_workers=4,
        val_idxs=val_idxs, suffix='.jpg', tfms = tfms, bs = bs)
    return data if sz>300 else data.resize(300, 'mp')
])

In [38]: learn = ConvLearner.pretrained(arch, data, precompute=True)
100% [██████████] 29/29 [00:38<00:00, 1.33s/it]

In [40]: learn.fit(0.01,5)
HBox(children=(IntProgress(value=0, description='Epoch', max=5), HTML(value='')))

epoch      trn_loss    val_loss    accuracy
0          1.74175  1.452699    0.476487
1          1.484502  1.442396    0.47787
2          1.362542  1.455849    0.47787
3          1.26602   1.445475    0.472337
4          1.17528   1.45434     0.480636

Out[40]: [array([1.45434]), 0.480636237815208]

In [41]: learn = ConvLearner.pretrained(arch, data, precompute=True)

In [42]: lrf=learn.lr_find()
HBox(children=(IntProgress(value=0, description='Epoch', max=1), HTML(value='')))
89% [██████████] 81/91 [00:01<00:00, 41.07it/s, loss=4.29]

In [43]: learn.sched.plot_lr()

```

```
In [43]: learn.sched.plot_lr()
```

The figure is a line plot with 'learning rate (log scale)' on the x-axis and 'validation loss' on the y-axis. The x-axis has major ticks at 10⁻⁴, 10⁻³, 10⁻², 10⁻¹, and 10⁰. The y-axis has major ticks at 2.3, 2.4, 2.5, 2.6, and 2.7. A blue line represents the validation loss. It starts at approximately 2.65 for a learning rate of 10⁻⁴, stays relatively flat until 10⁻³, then decreases steadily to a minimum of about 2.25 at a learning rate of 10⁻¹. After this minimum, the loss increases sharply, reaching approximately 2.7 at a learning rate of 10⁰.

```
In [44]: learn.sched.plot()
```

Learning Rate (log scale)	Validation Loss
10 ⁻⁴	~2.65
10 ⁻³	~2.60
10 ⁻²	~2.45
10 ⁻¹	~2.28
10 ⁰	~2.75

```
In [46]: learn.fit(0.01,3)

HBox(children=(IntProgress(value=0, description='Epoch', max=3), HTML(value='')))

epoch      trn_loss    val_loss    accuracy
0          1.021897    1.516875    0.472337
1          0.948211    1.51739    0.484786
```

```
In [48]: learn.fit(0.01,5, cycle_len = 1)
HBox(children=(IntProgress(value=0, description='Epoch', max=5), HTML(value='')))

epoch      trn_loss    val_loss    accuracy
0          1.420814    1.4487      0.472337
1          1.367504    1.424152    0.479253
2          1.356937    1.401127    0.478562
3          1.332531    1.386161    0.501383
4          1.307422    1.390915    0.487552

Out[48]: [array([1.39091]), 0.4875518671374763]

In [49]: learn.unfreeze()

In [50]: lr_rate = 0.01

In [51]: lr=np.array([lr_rate/6,lr_rate/3,lr_rate])

In [52]: learn.fit(lr, 3, cycle_len=1, cycle_mult=2)
HBox(children=(IntProgress(value=0, description='Epoch', max=7), HTML(value='')))

epoch      trn_loss    val_loss    accuracy
0          1.416926    1.281645    0.560858
1          1.276552    1.189014    0.598893
2          1.103698    1.149468    0.61065
3          1.107638    1.167215    0.598893
4          0.992899    1.160469    0.612725
5          0.852091    1.132254    0.615491
6          0.781523    1.129019    0.618257
```

```
In [52]: learn.fit(lr, 3, cycle_len=1, cycle_mult=2)

HBox(children=(IntProgress(value=0, description='Epoch', max=7), HTML(value='')))
```

epoch	trn_loss	val_loss	accuracy
0	1.416926	1.281645	0.568858
1	1.276552	1.189014	0.598893
2	1.183698	1.149468	0.61865
3	1.107638	1.167215	0.598893
4	0.992899	1.160489	0.612725
5	0.852091	1.132254	0.615491
6	0.781523	1.129819	0.618257

```
In [54]: accuracy_np(probs, y)
```

```
Out[54]: 0.6217158760719226
```

```
In [55]: preds = np.argmax(probs, axis=1)
probs = probs[:,1]
```

```
In [56]: from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y, preds)
```

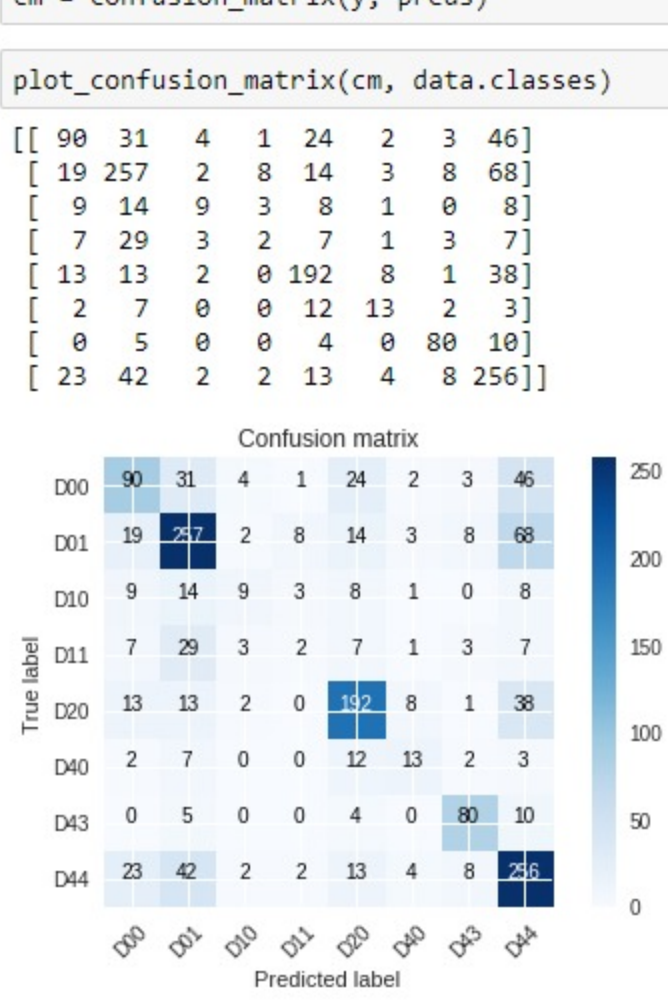
```
In [57]: plot_confusion_matrix(cm, data.classes)
```

	D00	D01	D10	D40	D43	D44
D00	90	24	4	1	24	2
D01	19	247	2	8	14	3
D10	9	14	9	3	8	1
D40	7	29	3	2	7	1
D43	13	13	2	0	192	8
D44	2	7	0	0	12	13

```
In [54]: accuracy_np(probs, y)
Out[54]: 0.6217150760719226

In [55]: preds = np.argmax(probs, axis=1)
         probs = probs[:,1]

In [56]: from sklearn.metrics import confusion
```



In []:


```
In [1]: %matplotlib inline
%reload_ext autoreload
%autoreload 2

In [24]: from fastai.conv_learner import *
from fastai.dataset import *

from pathlib import Path
import json
from PIL import ImageDraw, ImageFont
from matplotlib import patches, patheffects
torch.cuda.set_device(3)

In [ ]: PATH = Path('data/road-damage')
list(PATH.iterdir())

As well as the images, there are also annotations - bounding boxes showing where each object is. These were hand labeled. The original version were in XML, which is a little hard to work with nowadays, so we uses the more recent JSON version which you can download from this link.

You can see here how pathlib includes the ability to open files (amongst many other capabilities).

In [4]: trn_j = json.load((PATH/'road-damage.json').open())
trn_j.keys()

Out[4]: dict_keys(['images', 'type', 'annotations', 'categories'])

In [ ]: IMAGES,ANNOTATIONS,CATEGORIES = ['images', 'annotations', 'categories']
trn_j[IMAGES][:5]

In [8]: FILE_NAME,ID,IMG_ID,CAT_ID,BBOX = 'file_name','id','image_id','category_id','bbox'

cats = {o[ID]:o['name'] for o in trn_j[CATEGORIES]}
trn_fns = {o[ID]:o[FILE_NAME] for o in trn_j[IMAGES]}
trn_ids = {o[ID] for o in trn_j[IMAGES]}
```

```
In [13]: def trn_anno(o): return np.array([bb[1], bb[2], bb[3]-bb[1], bb[3]-bb[2]])

trn_anno = collections.defaultdict(lambda:[])
for o in trn_j[ANNOTATIONS]:
    if not o['ignore']:
        bb = o[BBOX]
        bb = hw_bb(bb)
        trn_anno[o[IMG_ID]].append((bb,o[CAT_ID]))

len(trn_anno)

Out[15]: 2581

In [14]: im_a = trn_anno[im0_d[ID]]; im_a

Out[14]: [(array([ 96, 155, 269, 350]), 7)]

In [15]: im0_a = im_a[0]; im0_a

Out[15]: (array([ 96, 155, 269, 350]), 7)

In [17]: trn_anno[17]

Out[17]: [(array([ 61, 184, 198, 278]), 15), (array([ 77, 89, 335, 482]), 13)]

Some libs take VOC format bounding boxes, so this lets us convert back when required:

In [21]: bb_voc = [155, 96, 196, 174]
bb_fastai = hw_bb(bb_voc)

In [24]: def bb_hw(a): return np.array([a[1],a[0],a[3]-a[1]+1,a[2]-a[0]+1])

In [20]: im = open_image(IMG_PATH/im0_d[FILE_NAME])

Matplotlib's plt.subplots is a really useful wrapper for creating plots, regardless of whether you have more than one subplot. Note that Matplotlib has an optional object-oriented API which I think is much easier to understand and use (although few examples online use it!)

In [20]: def show_img(im, figsize=None, ax=None):
    if not ax: fig,ax = plt.subplots(figsize=figsize)
    ax.imshow(im)
```

A simple but rarely used trick to making text visible regardless of background is to use white text with black outline, or visa versa. Here's how to do it in matplotlib.

```
In [21]: def draw_outline(o, lw):
    o.set_path_effects([patheffects.Stroke(
        linewidth=lw, foreground='black'), patheffects.Normal({})])

Note that * in argument lists is the splat operator. In this case it's a little shortcut compared to writing out b[-2],b[-1].

In [22]: def draw_rect(ax, b):
    patch = ax.add_patch(patches.Rectangle(b[:2], *b[-2:], fill=False, edgecolor='white', lw=2))
    draw_outline(patch, 4)

In [23]: def draw_text(ax, xy, txt, sz=14):
    text = ax.text(*xy, txt,
        verticalalignment='top', color='white', fontsize=sz, weight='bold')
    draw_outline(text, 1)

In [26]: def draw_in(im, ann):
    ax = show_img(im, figsize=(16,8))
    for b,c in ann:
        b = bb_hw(b)
        draw_rect(ax, b)
        draw_text(ax, b[:2], cats[c], sz=16)

In [ ]: ax = show_img(im)
b = bb_hw(im0_a[0])
draw_rect(ax, b)
draw_text(ax, b[:2], cats[im0_a[1]])

In [27]: def draw_idx(i):
    im_a = trn_anno[i]
    im = open_image(IMG_PATH/trn_fns[i])
    print(im.shape)
    draw_in(im, im_a)
```

```
In [20]: def get_lrg(b):
    if not b: raise Exception()
    b = sorted(b, key=lambda x: np.product(x[0][-2:]-x[0][:2]), reverse=True)
    return b[0]

In [26]: trn_lrg_anno = {a: get_lrg(b) for a,b in trn_anno.items()}

In [ ]: b,c = trn_lrg_anno[23]
b = bb_hw(b)
ax = show_img(open_image(IMG_PATH/trn_fns[23]), figsize=(5,10))
draw_rect(ax, b)
draw_text(ax, b[:2], cats[c], sz=16)

Now we have a dictionary from image id to a single bounding box - the largest for that image.

In [27]: (PATH/'tmp').mkdir(exist_ok=True)
CSV = PATH/'tmp/lrg.csv'

Often it's easiest to simply create a CSV of the data you want to model, rather than trying to create a custom dataset. Here we use Pandas to help us create a CSV of the image filename and class.

In [28]: df = pd.DataFrame({'fn': [trn_fns[o] for o in trn_ids],
    'cat': [cats[trn_lrg_anno[o][1]] for o in trn_ids]}, columns=['fn','cat'])
df.to_csv(CSV, index=False)

In [29]: f_model = resnet34
sz=224
bs=64

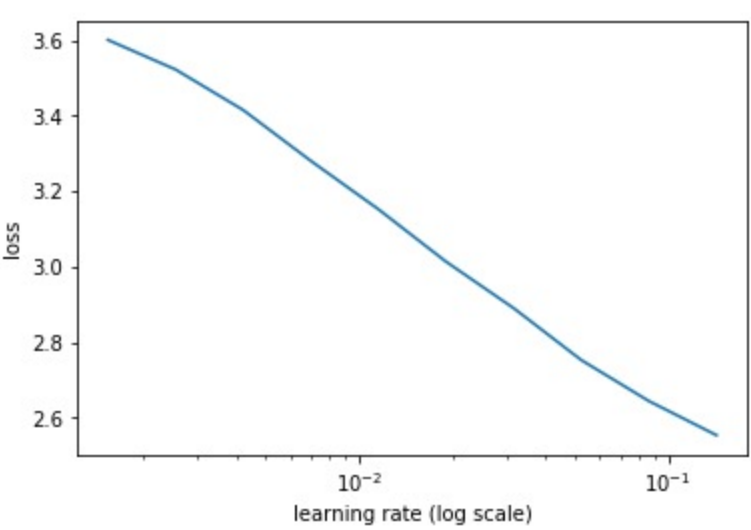
From here it's just like Dogs vs Cats!

In [30]: tfms = tfms_from_model(f_model, sz, aug_tfms=transforms_side_on, crop_type=CropType.NO)
md = ImageClassifierData.from_csv(PATH, JPEGs, CSV, tfms=tfms, bs=bs)

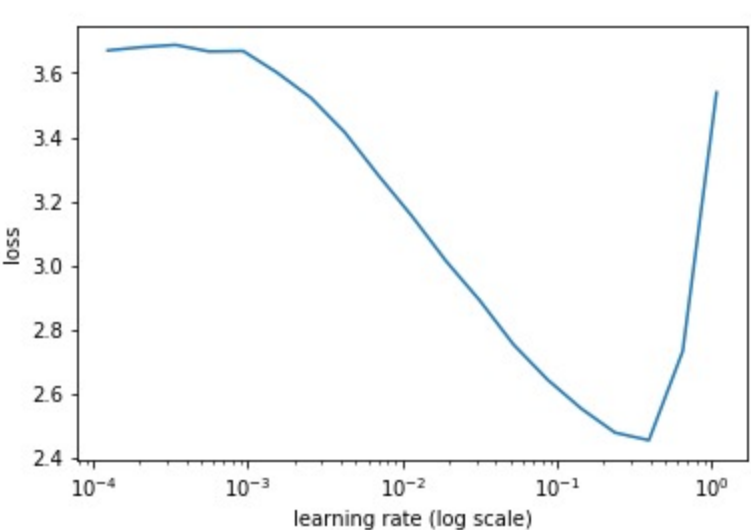
In [201]: x,y=next(iter(md.val_dl))
```

When you LR finder graph looks like this, you can ask for more points on each end:

```
In [36]: learn.sched.plot()
```



```
In [35]: learn.sched.plot(n_skip=5, n_skip_end=1)
```



```
In [43]: lr = 2e-2
```

```
Out[44]: [0.6443001, 0.80483774095773697]

In [45]: lrs = np.array([1r/1000,1r/100,1r])

In [46]: learn.freeze_to(23)
```

```
In [47]: learn.fit(lrs/5, 1, cycle_len=1)

A Jupyter Widget

epoch    trn_loss    val_loss    accuracy
0         0.609306    0.570568    0.821514
1         0.462856    0.574303    0.8128

Out[47]: [0.57553864, 0.82106370478868484]

In [48]: learn.unfreeze()

Accuracy isn't improving much - since many images have multiple different objects, it's going to be impossible to be that accurate.

In [49]: learn.fit(lrs/5, 1, cycle_len=2)

A Jupyter Widget

epoch    trn_loss    val_loss    accuracy
0         0.609306    0.570568    0.821514
1         0.462856    0.574303    0.8128

Out[49]: [0.57430345, 0.81280048191547394]

In [51]: learn.save('clas_one')

In [221]: learn.load('clas_one')

In [222]: x,y = next(iter(md.val_dl))
probs = F.softmax(predict_batch(learn.model, x), -1)
x,preds = to_np(x),to_np(probs)
preds = np.argmax(preds, -1)

In [ ]: fig, axes = plt.subplots(3, 4, figsize=(12, 8))
for i,ax in enumerate(axes.flat):
    ima=md.val_ds.denorm(x)[i]
    b = md.classes[preds[i]]
    ax = show_img(ima, ax=ax)
    draw_text(ax, (0,0), b)
plt.tight_layout()
```