# StreetView House Numbers: Pre-Processing

*Oliver Morris*

*26 July 2017*

## Summary

This notebook details the method used to prepare the data for a convolutional neural network (CNN) whose aim is to identify house numbers from photos. Those photos are made available in the SVHM dataset, which is from google streetview. The final model is 96% accurate, which is surprisingly good considering the 'entry level' PC and free software used.

The work is part of the capstone project for the Udacity 730 Tensorflow course. It follows the approach of the google research team in the paper Goodfellow, et al 2013.

The aim is process the images such that the digits within them have been extracted, are grayscale and are 32px x 32px. This size is small but compatible with the CNN model and hardware used. The hardware is a Windows 10 PC 64bit with Core i5-7500 (3.4GHz) 16Gb RAM, NVIDIA GeForce GTX1050Ti (4GB).

Steps are:

- Download and unzip images
- Crop images to bounding boxes around the digits
- Drop images which may mislead the CNN, eg very low quality
- Resize to 32x32 and set to greyscale
- Create train, validate and test tensors, ready for the CNN

Unusually, this notebook has been completed in R, the original course is in Python.

## Source Data

The source data is from Stanford. It is 250,000 images of house numbers taken from Google streetview. The data includes a list of those images and co-ordinates for boxes highlighting the digits within each image. This data processing will work with both the images and the boxes.

Beware, downloading and untarring this data may take hours. Over 250k files will be created, taking 4GB of disk space. Don't do this on your DropBox or OneDrive folder!

```
setwd("C:/Users/User/Documents/CNN_data/")
if(!file.exists("data")){
    dir.create("data")
}
download.file('http://ufldl.stanford.edu/housenumbers/train.tar.gz', "data/train.tar.gz")
download.file('http://ufldl.stanford.edu/housenumbers/test.tar.gz',  "data/test.tar.gz")
download.file('http://ufldl.stanford.edu/housenumbers/extra.tar.gz', "data/extra.tar.gz")

untar("data/train.tar.gz")
untar("data/test.tar.gz")
untar("data/extra.tar.gz")

library(R.matlab)
data <- readMat("train/digitStruct.mat")
```

## Unpack Hierarchical Files of Street View Data

Each image has a number of bounding boxes associated with it. Those boxes outline each digit in the image. The details of those boxes are in the .mat (Matlab 7.3, Hierarchical Data Format 5, aka HDF5) files which were untarred in the above chunk.

The Udacity 730 course code assistant, Hang Yao, has published a class in Python for extracting those files. The below Python chunk uses that class, then exports the resulting data as json to file, where it can be read by R in subsequent chunks.

```
import h5py
import os
import io
import json

os.chdir('C:\\Users\\OliverMorris\\Documents\\CNN_R\\')

# The DigitStructFile is just a wrapper around the h5py data.  It basically references
#    inf:              The input h5 matlab file
#    digitStructName   The h5 ref to all the file names
#    digitStructBbox   The h5 ref to all struc data
class DigitStructFile:
    def __init__(self, inf):
        self.inf = h5py.File(inf, 'r')
        self.digitStructName = self.inf['digitStruct']['name']
        self.digitStructBbox = self.inf['digitStruct']['bbox']

# getName returns the 'name' string for for the n(th) digitStruct.
    def getName(self,n):
        return ''.join([chr(c[0]) for c in self.inf[self.digitStructName[n][0]].value])

# bboxHelper handles the coding difference when there is exactly one bbox or an array of bbox.
    def bboxHelper(self,attr):
        if (len(attr) > 1):
            attr = [self.inf[attr.value[j].item()].value[0][0] for j in range(len(attr))]
        else:
            attr = [attr.value[0][0]]
        return attr

# getBbox returns a dict of data for the n(th) bbox.
    def getBbox(self,n):
        bbox = {}
        bb = self.digitStructBbox[n].item()
        bbox['height'] = self.bboxHelper(self.inf[bb]["height"])
        bbox['label'] = self.bboxHelper(self.inf[bb]["label"])
        bbox['left'] = self.bboxHelper(self.inf[bb]["left"])
        bbox['top'] = self.bboxHelper(self.inf[bb]["top"])
        bbox['width'] = self.bboxHelper(self.inf[bb]["width"])
        return bbox

    def getDigitStructure(self,n):
        s = self.getBbox(n)
        s['name']=self.getName(n)
        return s
```

```
# getAllDigitStructure returns all the digitStruct from the input file.
    def getAllDigitStructure(self):
        return [self.getDigitStructure(i) for i in range(len(self.digitStructName))]

# Return a restructured version of the dataset (one structure by boxed digit).
#
#   Return a list of such dicts :
#      'filename' : filename of the samples
#      'boxes' : list of such dicts (one by digit) :
#          'label' : 1 to 9 corresponding digits. 10 for digit '0' in image.
#          'left', 'top' : position of bounding box
#          'width', 'height' : dimension of bounding box
#
# Note: We may turn this to a generator, if memory issues arise.
    def getAllDigitStructure_ByDigit(self):
        pictDat = self.getAllDigitStructure()
        result = []
        structCnt = 1
        for i in range(len(pictDat)):
            item = { 'filename' : pictDat[i]["name"] }
            figures = []
            for j in range(len(pictDat[i]['height'])):
               figure = {}
               figure['height'] = pictDat[i]['height'][j]
               figure['label']  = pictDat[i]['label'][j]
               figure['left']   = pictDat[i]['left'][j]
               figure['top']    = pictDat[i]['top'][j]
               figure['width']  = pictDat[i]['width'][j]
               figures.append(figure)
            structCnt = structCnt + 1
            item['boxes'] = figures
            result.append(item)
        return result

## That the end of Hang Yao's class. Now use it to export data to file.
# Will be used by R in subsequent chunks
for datagroup in ['train','test','extra']:
  fin  = os.path.join(datagroup, 'digitStruct.mat')
  dsf  = DigitStructFile(fin)
  data = dsf.getAllDigitStructure_ByDigit()
  with io.open(datagroup+'/'+datagroup+'.json', 'w', encoding='utf-8') as outfile:
     outfile.write(json.dumps(data, ensure_ascii=False))
```

Having saved the python data as json files,load those into R

```
library(jsonlite)
library(data.table)

ConvertToDataTable <- function(BoxFileFromJSON){

  #approximately, how many boxes must we process?
  n <- length(unlist(BoxFileFromJSON, recursive = F))

  #this takes some time, so lets prep a progress bar
```

```
  pb <- txtProgressBar(min = 1, max = n, style = 3)

  #creat a template for the dataframe to store the boxes
  bboxes <- data.frame(filename = character(),
                       label    = numeric(),
                       width    = numeric(),
                       height   = numeric(),
                       top      = numeric(),
                       left     = numeric(),
                       stringsAsFactors = FALSE)

  eachrow <- 1
  bboxset <- 1
  # For each bboxset associated with a filename
  for (bboxset in 1:length(BoxFileFromJSON)) {

    bbox <- 1
    # For every bounding box associated with that filename
    for (bbox in 1:length(BoxFileFromJSON[[bboxset]]$boxes)) {

      # get the new row
      bboxes[eachrow,1] <- BoxFileFromJSON[[bboxset]]$filename
      bboxes[eachrow,2] <- as.integer(BoxFileFromJSON[[bboxset]]$boxes[[bbox]]$label)
      bboxes[eachrow,3] <- as.integer(BoxFileFromJSON[[bboxset]]$boxes[[bbox]]$width)
      bboxes[eachrow,4] <- as.integer(BoxFileFromJSON[[bboxset]]$boxes[[bbox]]$height)
      bboxes[eachrow,5] <- as.integer(BoxFileFromJSON[[bboxset]]$boxes[[bbox]]$top)
      bboxes[eachrow,6] <- as.integer(BoxFileFromJSON[[bboxset]]$boxes[[bbox]]$left)

      #This whole process takes a while, so lets monitor progress
      #Alternatively, maybe you can vectorise this function to make it faster...
      setTxtProgressBar(pb, eachrow)

      # increment the row
      eachrow<-eachrow+1

    }
  }

  # return the data as a Data Table
  as.data.table(bboxes)
}

train_bbox <- read_json("train/train.json", simplifyVector=F)
train_bbox_dt <- ConvertToDataTable(train_bbox)
write.csv(train_bbox_dt, "train/train.csv")
remove(train_bbox)

test_bbox  <- fromJSON("test/test.json",   simplifyVector=F)
test_bbox_dt <- ConvertToDataTable(test_bbox)
write.csv(test_bbox_dt, "test/test.csv")
remove(test_bbox)

extra_bbox <- fromJSON("extra/extra.json", simplifyVector=F)
```

```
extra_bbox_dt <- ConvertToDataTable(extra_bbox)
write.csv(extra_bbox_dt, "extra/extra.csv")
remove(extra_bbox)
```

Unify the data into one table. There's some data wrangling we need to do to all data sets, so best in one table.

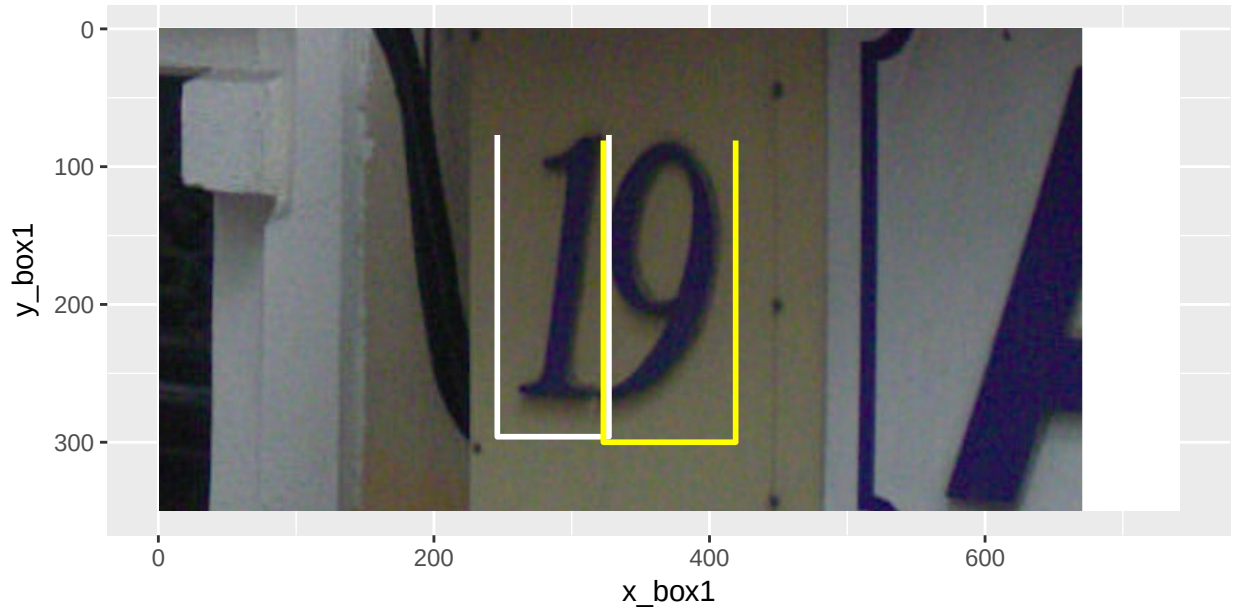
```
library(dplyr)
library(readr)

#if this work has already been done, then load from file...
if(file.exists("bbox_dt.csv")){
    bbox_dt<-read_delim("bbox_dt.csv", delim=" ")
}else{

  # apply folder names, so we know where to find each image
  train_bbox_dt <- train_bbox_dt %>% mutate(filename = paste0("train","/",filename))
  test_bbox_dt  <- test_bbox_dt  %>% mutate(filename = paste0("test" ,"/",filename))
  extra_bbox_dt <- extra_bbox_dt %>% mutate(filename = paste0("extra","/",filename))

  # unify the data sets
  bbox_dt <- rbind(train_bbox_dt, test_bbox_dt, extra_bbox_dt)

  #save this table to file as a checkpoint
  write_delim(bbox_dt, "bbox_dt.csv", delim=" ")
}
```

## Data Exploration

Let's explore the data. First step is to take a look at an example image with a box overlaid

```
library(imager)
library(grid)
library(ggplot2)

#Get 1.png for the box background
img <- load.image("train/1.png")
g <- rasterGrob(img, interpolate=TRUE)

# get x and y values for both boxes associated with 1.png, NB, not the width, height etc.
# note images such as these are referenced x,y where 0,0 is the top left corner, NOT bottom left.
# whereas ggplot will plot
imageexample <-  bbox_dt %>%
                  filter(filename=="train/1.png") %>%
                    mutate(filename, x0=left, y0=top, x1=left+width, y1=(top+height))


#arrange data for plotting the boxes
plotdat  <- data.frame(
                     x_box1=c(imageexample[1,]$x0,imageexample[1,]$x0,
                              imageexample[1,]$x1,imageexample[1,]$x1),
                     y_box1=c(imageexample[1,]$y0,imageexample[1,]$y1,
                              imageexample[1,]$y1,imageexample[1,]$y0),
```

```
                      x_box2=c(imageexample[2,]$x0,imageexample[2,]$x0,
                               imageexample[2,]$x1,imageexample[2,]$x1),
                      y_box2=c(imageexample[2,]$y0,imageexample[2,]$y1,
                               imageexample[2,]$y1,imageexample[2,]$y0))
ggplot(plotdat) +
  # draw the image
    annotation_custom(g, xmin=1, xmax=width(img), ymin=1, ymax=-height(img)) +
  # flip the y scale, so that 0,0 is top left. this is the standard for images
    scale_y_continuous(trans=scales::reverse_trans()) +
  # plot the two boxes
    geom_line(color='white', size=1, aes(x=x_box1,y=y_box1)) +
    geom_line(color='yellow',size=1, aes(x=x_box2,y=y_box2)) +
  # force plot from 0,0
    expand_limits(x=c(0,width(img)), y=c(0, height(img))) +
  # force x pixel size = y pixel size (square pixels, not rectangular)
    coord_fixed()
```




### Confirm Reliability of Data

This raises a good question, is the left most box always the first in sequence in the table? The data asks us to assume this is the case, but provides no evidence. Let's check that assumption...

```
# get the sequence of digits, as presented in the original data
bbox_dt <- tibble::rownames_to_column(bbox_dt, "seq_presented")
```

```
# get the sequence of digits, as calculated by assuming the leftmost is first in sequence
bbox_dt_lab <- bbox_dt %>%
                #calculate row number in order of x position
                arrange(filename,left) %>%
                  group_by(filename) %>%
                    mutate(seq_x = dplyr::row_number(left)) %>%
                      ungroup() %>%
                      #calculate row number in order of original file
                        arrange(seq_presented) %>%
                          group_by(filename) %>%
                            mutate(seq_file = dplyr::row_number(seq_presented)) %>%
                              ungroup()

# we don't need seq_presented any more...
bbox_dt_lab$seq_presented <- NULL

# View those where the sequence presented does not match the sequence assuming first is leftmost.
bbox_dt_lab %>% filter(seq_file != seq_x) %>%
                    arrange(filename, seq_file) %>%
                      select(filename, label, seq_file, seq_x)
```

### Drop Unreliable Data then Tidy into One Digit Per Column

There are only 150 digits, 56 images, where the sequences don't match. Out of hundreds of thousands of images we can afford to simply remove these from the training and testing data. We also need to tidy the digit sequence into one digit per column.

```
library(tidyr)

#load checkpoint, if exists
if(file.exists("bbox_dt_lab.csv")){
  bbox_dt_lab <- read_delim("bbox_dt_lab.csv",delim=",")
}else{

  # There are only 53 such images in the entire data set.
  # Visual inspection showed these are probably errors of some sort, they will simply be dropped.
  bbox_dt_lab <- bbox_dt_lab %>% filter(seq_x == seq_file)

  #we don't need the file sequence any more
  bbox_dt_lab$seq_file <- NULL

  # Spread the data so that each label has its own column, this will prove useful for later analysis
  # NB, the number 10 is being used to denote a ZERO. So, we designate a 'blank' as '11'.
  bbox_dt_lab <- spread(bbox_dt_lab, key=seq_x, value=label, fill=11)


  #save checkpoint
  write_delim(bbox_dt_lab,"bbox_dt_lab.csv",delim=",")
}
# let's see our new table's format
head(bbox_dt_lab %>% arrange(filename))
```

```
## # A tibble: 6 x 11
```

```
##       filename width height   top  left   `1`   `2`   `3`   `4`   `5`
##          <chr> <int>  <int> <int> <int> <int> <int> <int> <int> <int>
## 1  extra/1.png    38     56    70    24     4    11    11    11    11
## 2  extra/1.png    36     56    41    55    11     7    11    11    11
## 3  extra/1.png    47     56    23    79    11    11     8    11    11
## 4 extra/10.png    18     25     5     5     4    11    11    11    11
## 5 extra/10.png    13     25     7    25    11     4    11    11    11
## 6 extra/10.png    12     25     7    40    11    11     4    11    11
## # ... with 1 more variables: `6` <int>
```

### Coalesce Digit Boxes into One Box for Entire House Number

The x,y,width, height format for each box within an image is cumbersome for our use. We need list of x's and y's for the 'coalesced' box, i.e. the grouping of the boxes. This results in a new dataframe, one row per image (not one row per box): filename, x0, y0, x1, y1, labels (list), num_digits

```
getCoalescedBoxes <- function(dataset){
  dataset_coal <- dataset %>%
                  mutate(x0box=left, y0box=top, x1box=left+width, y1box=top+height) %>%
                  group_by(filename) %>%
                  summarise(x0 = as.integer(min(x0box)),
                            y0 = as.integer(min(y0box)),
                            x1 = as.integer(max(x1box)),
                            y1 = as.integer(max(y1box)),
                            DigitCount = n(),
                            # We can use min() to sift the label from the 11's (ie, 'no digits')
                            LabelA = min(`1`, na.rm = TRUE),
                            LabelB = min(`2`, na.rm = TRUE),
                            LabelC = min(`3`, na.rm = TRUE),
                            LabelD = min(`4`, na.rm = TRUE),
                            LabelE = min(`5`, na.rm = TRUE),
                            LabelF = min(`6`, na.rm = TRUE)
                            )
}

#load checkpoint, if exists
if(file.exists("bbox_dt_lab_coal.csv")){
  bbox_dt_lab_coal <- read_delim("bbox_dt_lab_coal.csv",delim=",")
}else{
  bbox_dt_lab_coal <- getCoalescedBoxes(bbox_dt_lab)
}

# Save checkpoint
write_delim(bbox_dt_lab_coal, "bbox_dt_lab_coal.csv", delim=",")

# Take a peek at the data
head(bbox_dt_lab_coal %>% arrange(filename))
```

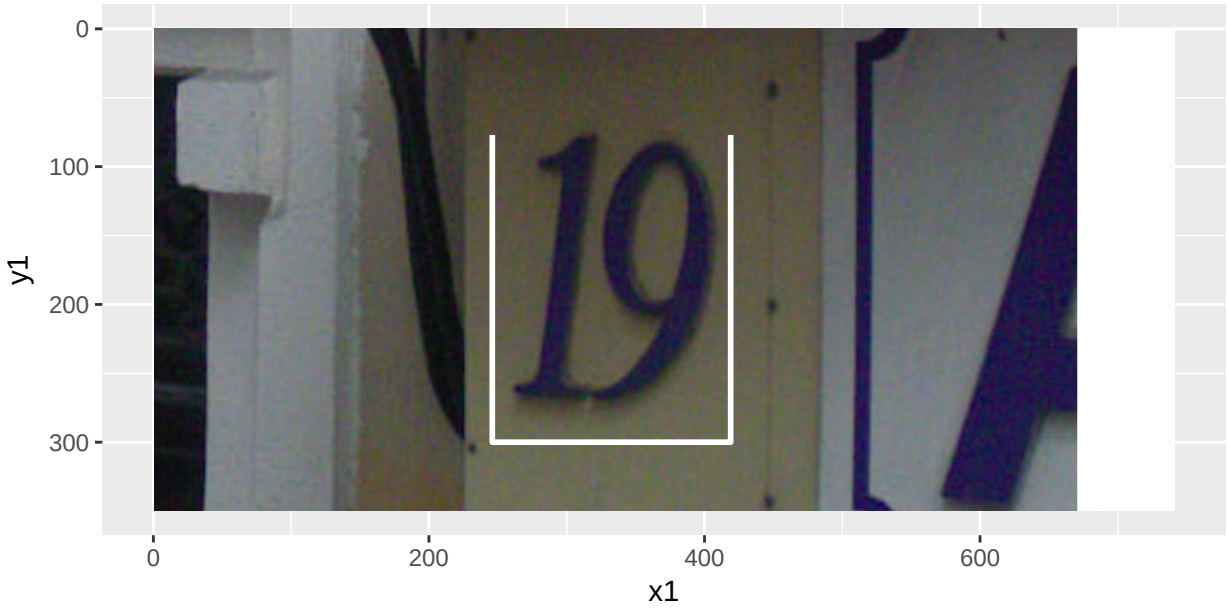
```
## # A tibble: 6 x 13
##      X1         filename    x0    y0    x1    y1 DigitCount LabelA LabelB
##   <int>            <chr> <int> <int> <int> <int>      <int>  <int>  <int>
## 1     1      extra/1.png    24    23   126   126          3      4      7
## 2     2     extra/10.png     5     5    52    32          3      4      4
## 3     3    extra/100.png     3     1    46    30          3      5      3
```

```
## 4     4   extra/1000.png     9     7    36    40          2      2      6
## 5     5  extra/10000.png     8     9    41    35          3      2     10
## 6     6 extra/100000.png    33    10    63    29          3      1      5
## # ... with 4 more variables: LabelC <int>, LabelD <int>, LabelE <int>,
## #   LabelF <int>
```

### Plot Completed Examples

Let's plot the coalesced box for 1.png, as an example...

```
img <- load.image("train/1.png")
g   <- rasterGrob(img, interpolate=TRUE)

#get x and y values for each box for 1.png, not the width height etc.
imageexample <-  bbox_dt_lab_coal %>%
                  filter(filename=="train/1.png")

#arrange data for plotting the boxes
plotdat  <- data.frame(x1=c(imageexample$x0,imageexample$x0,
                            imageexample$x1,imageexample$x1),
                       y1=c(imageexample$y0,imageexample$y1,
                            imageexample$y1,imageexample$y0))

#plot image with box overlay
ggplot(plotdat) +
    annotation_custom(g, xmin=1, xmax=width(img), ymin=1, ymax=-height(img)) +
    scale_y_continuous(trans=scales::reverse_trans()) +
    geom_line(color='white', size=1, aes(x=x1,y=y1)) +
    expand_limits(x=c(0,width(img)), y=c(0, height(img))) +
    coord_fixed()
```




## Get Image Sizes

Our table is missing some crucial information, the size of the images. We'll cycle through the table using the imager package to get image size (height/width) and append that to the table.

```
getImageSize <- function(imageLocn){
  image <- load.image(imageLocn)
  dims  <- dim(image)[1:2]
  paste0(dims[1],",",dims[2]) #returns dims as width [space] height, all in one char string.
  }

getImageSize_vec <- Vectorize(getImageSize)

# this is a time consuming process, so load checkpoint, if exists
if(file.exists("bbox_dt_lab_coal_dims.csv")){
  bbox_dt_lab_coal_dims <- read_delim("bbox_dt_lab_coal_dims.csv",delim=",")
}else{

  # prepare a new dataframe for the image dimensions
  # this implies 248,000+ file reads, so we will read each image only once to get width and height
  # both values will be written into one column for speed, a subsequent steps separates columns out
  bbox_dt_lab_coal <- bbox_dt_lab_coal %>% mutate(dims = getImageSize_vec(filename))

  # split the dims columns into width and height
  bbox_dt_lab_coal_dims <- bbox_dt_lab_coal %>%
```

```
                            separate(dims, into = c("img_width", "img_height"), sep = ",")
  bbox_dt_lab_coal_dims$img_width  <- as.integer(bbox_dt_lab_coal_dims$img_width)
  bbox_dt_lab_coal_dims$img_height <- as.integer(bbox_dt_lab_coal_dims$img_height)

  #We've inadvertantly acquired a couple of columns of rownames, let's drop those
  bbox_dt_lab_coal_dims <- bbox_dt_lab_coal_dims[,3:16]

  # save checkpoint
  write_delim(bbox_dt_lab_coal_dims,"bbox_dt_lab_coal_dims.csv",delim=",")
}

head(bbox_dt_lab_coal_dims)
```

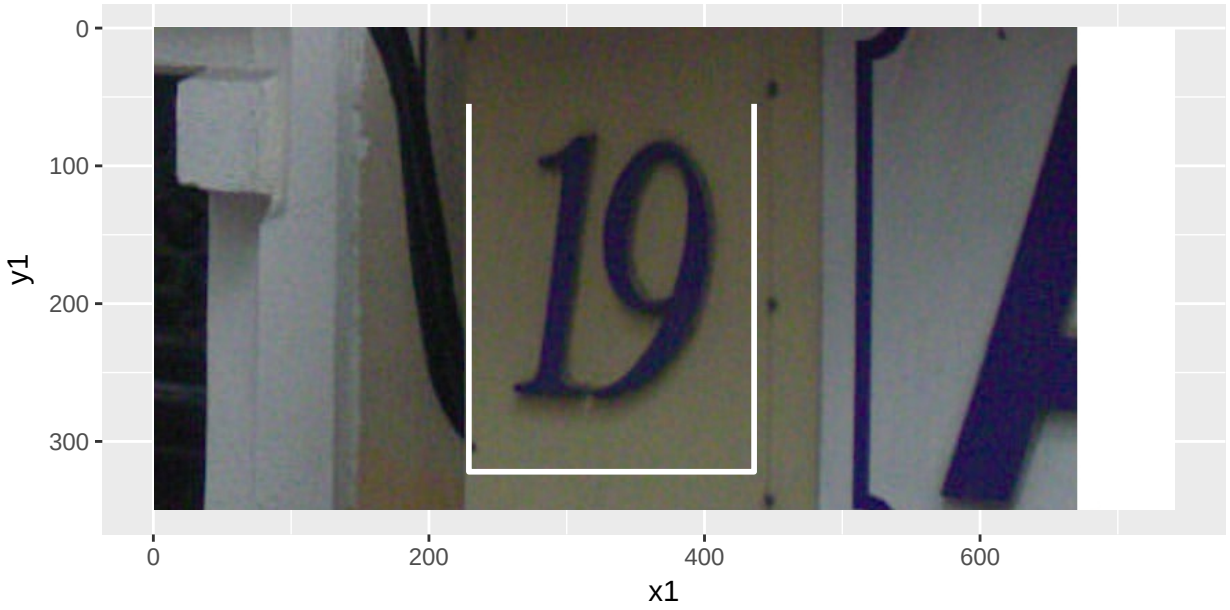
```
## # A tibble: 6 x 16
##      X1         filename    x0    y0    x1    y1 DigitCount LabelA LabelB
##   <int>            <chr> <int> <int> <int> <int>      <int>  <int>  <int>
## 1     1      extra/1.png    24    23   126   126          3      4      7
## 2     2     extra/10.png     5     5    52    32          3      4      4
## 3     3    extra/100.png     3     1    46    30          3      5      3
## 4     4   extra/1000.png     9     7    36    40          2      2      6
## 5     5  extra/10000.png     8     9    41    35          3      2     10
## 6     6 extra/100000.png    33    10    63    29          3      1      5
## # ... with 7 more variables: LabelC <int>, LabelD <int>, LabelE <int>,
## #   LabelF <int>, img_width <int>, img_height <int>, filename_c <chr>
```

## Expand Image Boxes

The image box looks a little tight around the digits, so we will expand the bounding box by approx 20% in each direction (20% in x and 20% in y).

```
# ensure all libraries available, as required
library(ggplot2)
library(grid)
library(imager)
library(dplyr)

#expand dims by 10% of wdith for x0 and same for x1 (20% total in x direction). Same for y direction
bbox_dt_lab_coal_wide <-bbox_dt_lab_coal_dims %>%
                        mutate(x0_new=as.integer(pmax(round(x0-(x1-x0)*0.1, digits=0),0)),
                               x1_new=as.integer(pmin(round(x1+(x1-x0)*0.1, digits=0),img_width)),
                               y0_new=as.integer(pmax(round(y0-(y1-y0)*0.1, digits=0),0)),
                               y1_new=as.integer(pmin(round(y1+(y1-y0)*0.1, digits=0),img_height)))

#get x and y values for each box for 1.png, not the width height etc.
imageexample <-  bbox_dt_lab_coal_wide %>%
                 filter(filename=="train/1.png")

#arrange data for plotting the boxes
plotdat  <- data.frame(x1=c(imageexample$x0_new,imageexample$x0_new,
                            imageexample$x1_new,imageexample$x1_new),
                       y1=c(imageexample$y0_new,imageexample$y1_new,
                            imageexample$y1_new,imageexample$y0_new))
#plot
```

```
ggplot(plotdat) +
    annotation_custom(g, xmin=1, xmax=width(img), ymin=1, ymax=-height(img)) +
    scale_y_continuous(trans=scales::reverse_trans()) +
    geom_line(color='white', size=1, aes(x=x1,y=y1)) +
    expand_limits(x=c(0,width(img)), y=c(0, height(img))) +
    coord_fixed()
```




## Crop & Grayscale

Colour does not give us any information about numbers, so we should greyscale the images to prevent spurious colour associations, then crop to the new expanded bounding box.

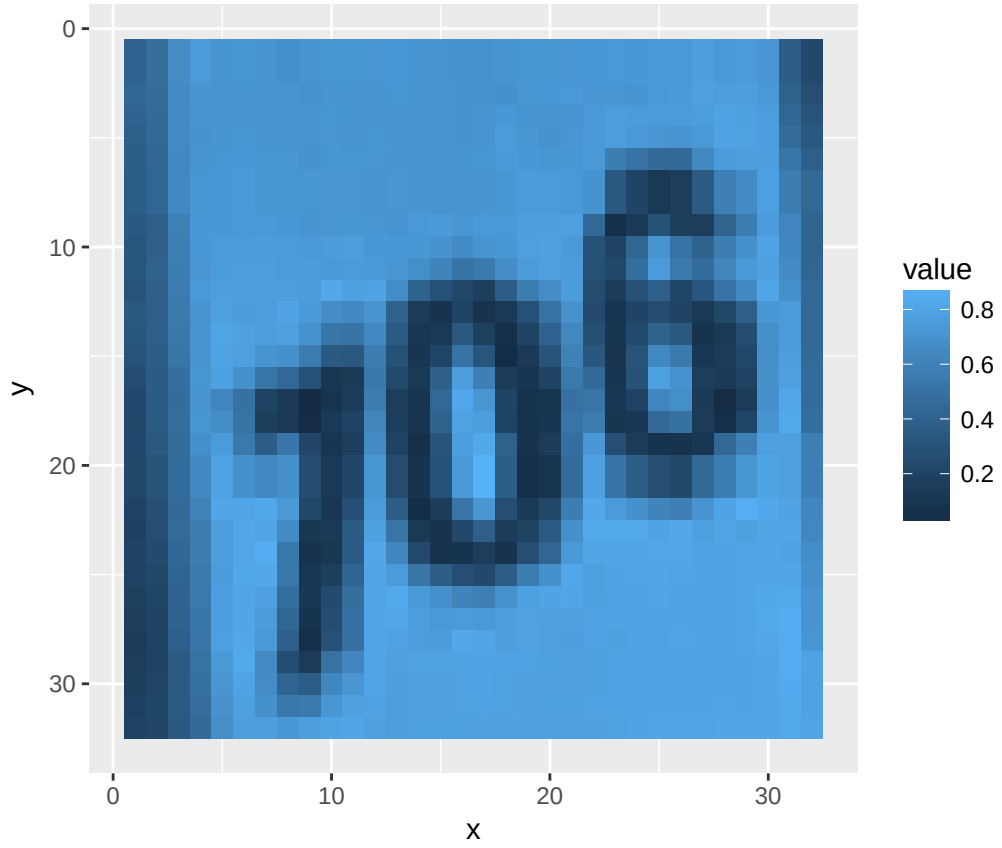
```
# add a new column for the new processed location\filename
bbox_dt_lab_coal_wide_crop <- mutate(bbox_dt_lab_coal_wide, filename_c = as.character(NA))

# create function to do processing of image
GrayCropAndScale <- function(params){

  # function turns image to grayscale, crops to bounding box,
  # then scales image into standard size of 32x32
  image <- load.image(as.character(params$filename))

  # convert cimg object to grayscale
  image <- grayscale(image)
```

```
# get pixels as x,y (where 0,0 is top left) from cimg object,
# done simply by converting to dataframe
image <- as.data.frame(image)

# to get the crop, we simply select the pixels with x,y such that x0<x<x1 and y0<y<y1
# We subtract or add 1 from the new dims so that x and y are never 0, must be 1 at a minimum.
cropped <- image %>%
  filter(between(x, params$x0_new-1, params$x1_new+1)) %>%
  filter(between(y, params$y0_new-1, params$y1_new+1))

# reset x, y so that top left is 0,0
# (the original 0,0 likely got cropped off the image in the above step)
cropped <- cropped %>%
  transmute(x = x-pmax((params$x0_new-2),0),
            y = y-pmax((params$y0_new-2),0),
            value=value)

# the grayscale values tend to be very small and get truncated during the save operation
# so, we rescale values to values betwen 0 and 1.
cropped$value <- (cropped$value-min(cropped$value))/max(cropped$value)

# before we resize we must stretch the image so width=height, i.e. square it.
width  <- params$x1_new - params$x0_new
height <- params$y1_new - params$y0_new

if(width==height){
  square <- as.cimg(cropped)
}else{
  if(width>=height){
    square <- suppressWarnings(resize(as.cimg(cropped),
                                      size_x = width,
                                      size_y = -width/height*100))
  }else{
    square <- suppressWarnings(resize(as.cimg(cropped),
                                      size_x = -height/width*100,
                                      size_y = height))
  }
}

# Having squared it we can safely resize to 32px x 32px,
# as will be required by our Convolutional Neural Net
resized <- suppressWarnings(resize(square, size_x = 32L, size_y = 32L, interpolation_type = 1))

# save to subfolder. Format as png is set by file name.
filen <- paste0("cropped\\",
                strsplit(params$filename, "/")[[1]][1],
                "_",
                strsplit(params$filename, "/")[[1]][2])

save.image(resized, filen)

# return only the new file location
return(paste0(filen))
```

```
}

# loop through all images the hard way, with a for loop
# This process takes hours on a PC as there are over 240,000 image files to be processed
# so check if you've already done this step, else go watch some telly...

if(file.exists("bbox_dt_lab_coal_wide_crop.csv")){
  bbox_dt_lab_coal_wide_crop <- read_delim("bbox_dt_lab_coal_wide_crop.csv",delim=",")
}else{
  rows<- nrow(bbox_dt_lab_coal_wide)
  pb  <- txtProgressBar(min = 1, max = rows, style = 3)
  for (eachrow in 1:rows) {
    bbox_dt_lab_coal_wide_crop$filename_c[eachrow] <- GrayCropAndScale(#params
                                                        select(bbox_dt_lab_coal_wide_crop,
                                                               filename,
                                                               x0_new,
                                                               x1_new,
                                                               y0_new,
                                                               y1_new)[eachrow]
                                                        )
    setTxtProgressBar(pb, eachrow)
  }
  #create checkpoint
  bbox_dt_lab_coal_wide_crop <- write_delim(bbox_dt_lab_coal_wide_crop,
                                            "bbox_dt_lab_coal_wide_crop.csv",delim=",")
}
```

**Lets look at an example grayscale, cropped, 32px x 32px image**

```
image <- load.image(as.character(bbox_dt_lab_coal_wide_crop$filename_c[1]))

ggplot(as.data.frame(image),aes(x,y)) +
  geom_raster(aes(fill=value)) +
  scale_y_continuous(trans=scales::reverse_trans()) +
  coord_fixed()
```




## Data cleaning

Are there any images where the DigitCount does not match the number of labels we have? – if so then the image will confuse training and should be discarded

```
DropDueLabelError <-
             bbox_dt_lab_coal_wide_crop %>%
             rowwise() %>%
             filter(DigitCount!=sum(ifelse(c(LabelA,LabelB,LabelC,LabelD,LabelE,LabelF)!=11,1,0)))

bbox_dt_lab_coal_wide_crop <- anti_join(bbox_dt_lab_coal_wide_crop,
                                        DropDueLabelError, by = "filename")
```

### Remove Low Resolution Images

The above chunk shows obvious label errors, although by visual inspection there are a few. We'll have to tolerate some errors in the ground truth, this was also encountered by the Google team.

Are there any images below a minimum acceptable resolution after cropping (but before resizing to 32x32)? At least 64px per digit (8px x 8px)? – if so then the image is of such low quality that it may confuse training, and should be discarded

```
DropDueResolution <- bbox_dt_lab_coal_wide_crop %>%
                      mutate(cropped_width=x1_new - x0_new, cropped_height = y1_new-y0_new) %>%
                        filter(cropped_width*cropped_height < 64*DigitCount) %>%
```

```
                          select(filename, filename_c, DigitCount, cropped_width, cropped_height)

bbox_dt_lab_coal_wide_crop <- anti_join(bbox_dt_lab_coal_wide_crop,
                                        DropDueResolution, by = "filename")
```

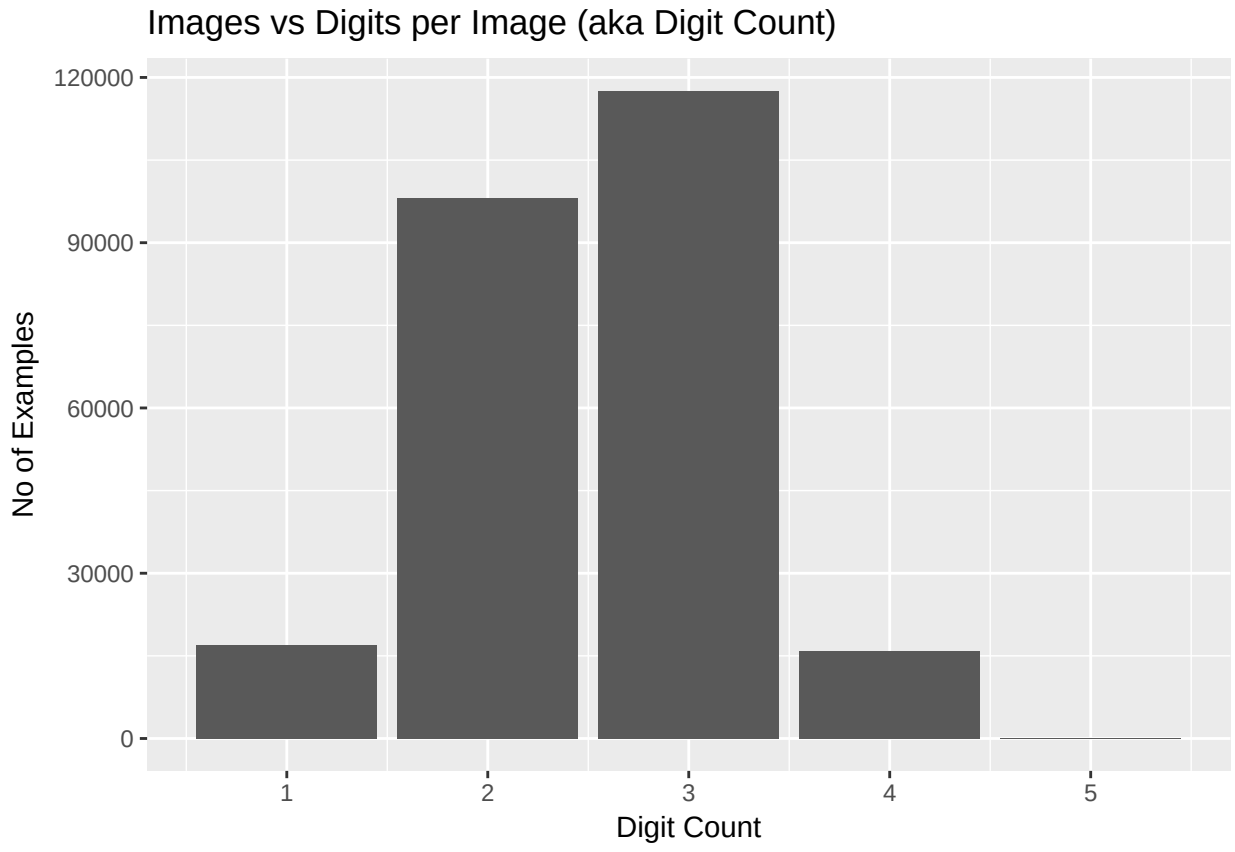
## Distribution of Data

Finally we get a chance to explore clean and processed data, understand the distribution of examples (aka images) from which we will ask the convolutional neural net to learn.

### Frequency of DigitCount

What is the frequency of each DigitCount? – do we have enough of each type of image to train a model?

```
dat_DigitCountFrequency <- bbox_dt_lab_coal_wide_crop  %>%
                            group_by(DigitCount) %>%
                              dplyr::summarise(Freq=n())

ggplot(data=dat_DigitCountFrequency, aes(x=DigitCount, y=Freq)) +
  geom_bar(stat="identity") +
  scale_x_continuous(breaks=c(1,2,3,4,5,6)) +
  labs(title="Images vs Digits per Image (aka Digit Count)") +
  labs(x="Digit Count", y="No of Examples")
```



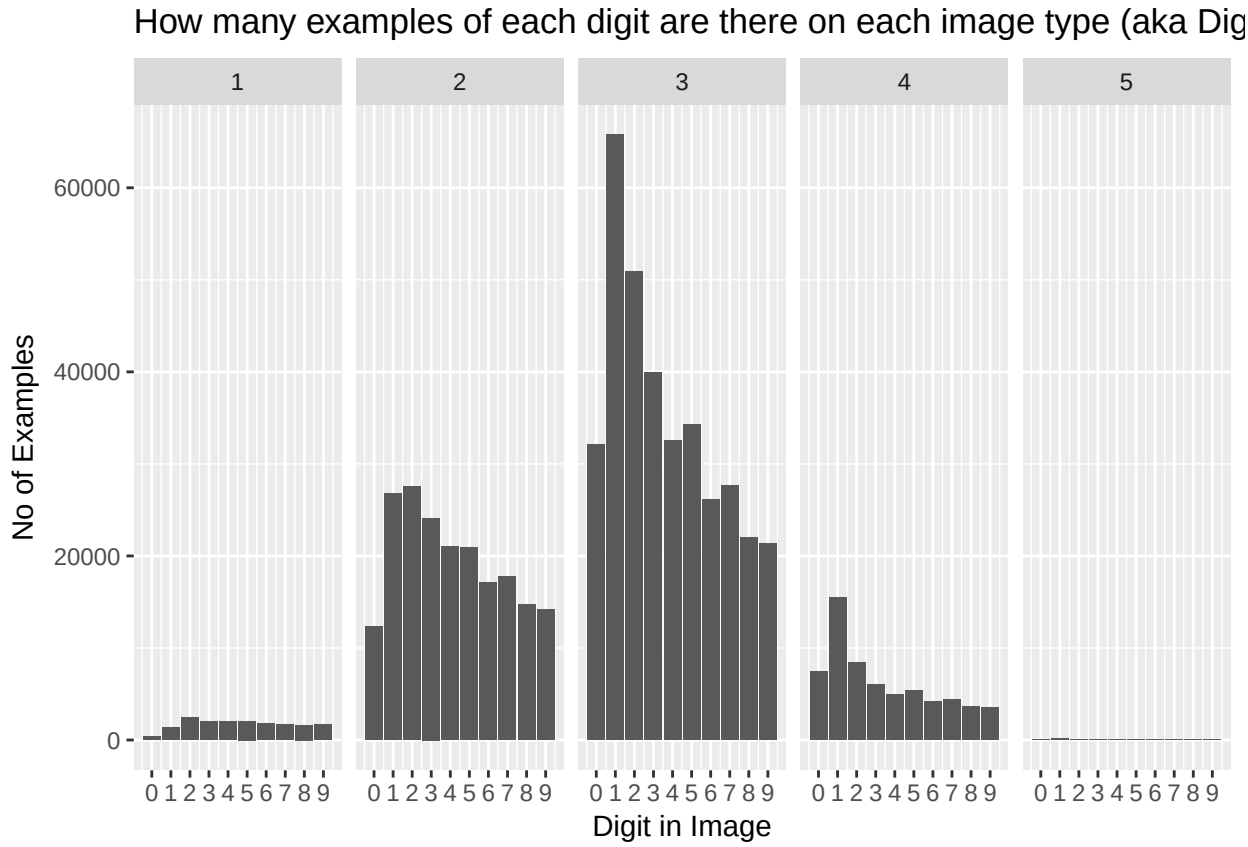
There is only 1 example of 6 digit house numbers. We should not attempt to train for 6 digits, so drop this example from the list. There are only 123 examples of 5 digit house numbers, so should expect poor model training for 5 digits. The data will be left in the model and we will see how well it performs on so few examples.

```
SixDigit <- bbox_dt_lab_coal_wide_crop %>% filter(DigitCount==6)
bbox_dt_lab_coal_wide <- anti_join(bbox_dt_lab_coal_wide_crop, SixDigit, by = "filename")

#overwrite checkpoint
write_delim(bbox_dt_lab_coal_wide_crop,"bbox_dt_lab_coal_wide_crop.csv",delim=",")
```

### Frequency of Digits

What is the frequency of digits for each type of DigitCount? – how well is each digit represented in the sample?

```
dat_LabelFreqPerDigitCount <- bbox_dt_lab_coal_wide_crop %>%
                                select(DigitCount, LabelA,LabelB,LabelC,LabelD,LabelE,LabelF) %>%
                                  gather(LabelName, Label, -DigitCount)

pltdat_LabelFreqPerDigitCount <- dat_LabelFreqPerDigitCount %>%
                                  group_by(DigitCount, Label) %>%
                                    dplyr::summarise(Qty=n()) %>%
                                      mutate(LabelName=ifelse(Label==10,0,Label)) %>%
                                        filter(LabelName!=11)

ggplot(data=pltdat_LabelFreqPerDigitCount, aes(x=LabelName, y=Qty)) +
  geom_bar(stat="identity") +
  scale_x_continuous(breaks=c(0,1,2,3,4,5,6,7,8,9)) +
  labs(title="How many examples of each digit are there on each image type (aka Digit Count)?") +
  labs(x="Digit in Image", y="No of Examples") +
  facet_grid(. ~ DigitCount, scales = "free")
```




There is a clear preference for images with three digits. Within all types of image there is a preference for digits 1,2 and 3. This is not surprising given that all streets must start with low numbers and may not be long enough to reach high numbers, so the test data will reflect real life data. No action results from this.

### Sample for Train & Test Data Sets

Final step is to select our own randomised samples for train, validation and test. The original data provided 13,000+ images for testing, so 13,000 images will be provided for test, albeit randomly sampled from each of the original groups. So, what to do with the 'extra' set, which is the bulk of images? The majority would be best employed in training, with a rest being a validation set. Validation only needs to be half as numerous as testing, so 6500 examples will suffice.

```
#seek checkpoint
if(file.exists("bbox_dt_lab_coal_wide_samp.csv")){
  bbox_dt_lab_coal_wide_samp <- read_delim("bbox_dt_lab_coal_wide_samp.csv", delim=",")
}else{

  # create a new column for flagging which image has been assigned to which set
  bbox_dt_lab_coal_wide_samp <- bbox_dt_lab_coal_wide_crop %>% mutate(sampleset=as.character(NA))

  ## TEST
  # identify 13,000 random samples for the test set
  set.seed(201707)
  filenames_test <- as.data.table(sample_n(bbox_dt_lab_coal_wide_samp, 13000,
                                           replace = FALSE)$filename_c)
```

```
  # update the main list of all images with a flag to show which have been assigned to 'test'
  bbox_dt_lab_coal_wide_samp <- bbox_dt_lab_coal_wide_samp %>%
                                mutate(sampleset = replace(sampleset,
                                                           (filename_c %in% filenames_test$V1),
                                                           "test"))
  ## VALIDATE
  # identify 6,500 random samples, which have not already been assigned to 'test', for validation
  set.seed(201707)
  filenames_validate <- as.data.table(sample_n((bbox_dt_lab_coal_wide_samp %>%
                                                  filter(is.na(sampleset))),
                                               6500, replace = FALSE)$filename_c)

  # update the main list of all images with a flag to show which have been assigned to 'validate'
  bbox_dt_lab_coal_wide_samp <- bbox_dt_lab_coal_wide_samp %>%
                                mutate(sampleset = replace(sampleset,
                                                           (filename_c %in% filenames_validate$V1),
                                                           "validate"))
  ## TRAIN
  # Everything else must be for 'train'
  bbox_dt_lab_coal_wide_samp <- bbox_dt_lab_coal_wide_samp %>%
                                mutate(sampleset = replace(sampleset,
                                                           is.na(sampleset),
                                                           "train"))

  # For the sake of the next code chunk (creating tensors), let's apply a clear order to the table
  # so we can always track back from the forthcoming tensor to the master table.
  bbox_dt_lab_coal_wide_samp <- bbox_dt_lab_coal_wide_samp %>%
                                  arrange(sampleset, filename_c) %>%
                                    mutate(sequence = seq(1:nrow(bbox_dt_lab_coal_wide_samp)))
}

# report sample sizes
bbox_dt_lab_coal_wide_samp %>%
  group_by(sampleset) %>%
    dplyr::summarise(count=n())
```

```
## # A tibble: 3 x 2
##   sampleset  count
##       <chr>  <int>
## 1      test  13000
## 2     train 229260
## 3  validate   6500
```

## Preparing the Data for the Tensorflow Graph

To recap, a table has been created which is a master list of all the images, their file location, original size and to which sample set they have been assigned.

For the neural net we need the x,y,grayscale value for each image, and thats it. So, the tensors (as we will call them) will need to have these shapes:

- test = [ 13000, 32, 32, 1]
- train = [229260, 32, 32, 1]
- validate = [ 6500, 32, 32, 1]

We also need the labels, A thru E (don't need F, 6 digit images have been excluded). The first label is the quantity of digits on the image, aka DigitCount, numbers 1 to 5. Subsequent labels are the digits in the image, where each digit is 0-11. 10 represents 0, 11 represents blank. So, thats 6 labels for each image.

Although the following code reads all 240,000+ 32x32 png images, the process is surprisingly quick. However, it requires a machine with approx 16GB memory. The largest file, x_train, is approx 4Gb on disk. NB, it helps to keep a Rdata snapshot of variables, as it loads much more quickly from binary.

## IMPORTANT: Scale the Grayscale Values

During the process the image's grayscale values are scaled so that mean=0 and SD=1 for each image. This is a common preprocessing task for machine learning, ensuring all input data is on the same scale. ReLu's (the activiation method to be used) function well with weights trained to data centered at zero, because unused weights tend to get clipped, as is the intention with ReLu.

If the data is not scaled then final model accuracy will suffer. The model may even fail to train due to weights 'exploding', which initially happened in this project, prior to input data being scaled.

```
library(reshape2)
library(readr)
# First convert the observations (aka 'x', or the 'images') for saving to file

CreateSamplesetTensor <- function(data, subset_name){
  # filter down to subset
  subset <- data %>%
              filter(sampleset == subset_name) %>%
                 arrange(sequence)

  #prepare for loop to convert images to list of matrices
  rows <- nrow(subset)
  pb   <- txtProgressBar(min = 1, max = rows, style = 3)
  imagetensor <- list()

  for (eachrow in 1:rows){

  # load image
    image <- load.image(as.character(subset$filename_c[eachrow]))
  # get pixels as x,y and grayscale value
    image <- as.data.frame(image)
  # scale each image, (mean=0, sd=1)
    image$value <- scale(image$value)
  # Values currently stored to 8 decimal places, eg 0.12345678, using more memory than necessary
  # 3 decimal places, eg 0.123, are sufficient, it moves the mean (a little) but smaller file
    image$value <- round(image$value,3)
  # Convert to 32x32 matrix
    m <- matrix(image$value, nrow = 32, byrow = TRUE)
  # Append to list of image matrices
    imagetensor[[eachrow]] <- m
    setTxtProgressBar(pb, eachrow)

  }

  # use the melt() method from reshape2 to help save the data in an intelligible format
  # the gather() method from tidyr is comparable but does not take lists or matrices.
```

```
  imagetensor_write <- melt(imagetensor)
  colnames(imagetensor_write) <- c("y","x","value","Sequence")
    #NB 'sequence' now runs 1-n for all sets, so there are three sequence=1.

  # Use the readr package to write the file to disk as efficiently as possible, machine needs 16Gb
  # using the base write.csv() function will result in out-of-memory errors on machines <24Gb
  write_delim(imagetensor_write, paste0("imagetensor_",subset_name,".txt"), delim = " ")

  #return something
  print(paste0(subset_name, " images: Done"))
  return(imagetensor)
}

x_validate  <- CreateSamplesetTensor(bbox_dt_lab_coal_wide_samp, "validate")
x_test      <- CreateSamplesetTensor(bbox_dt_lab_coal_wide_samp, "test")
x_train     <- CreateSamplesetTensor(bbox_dt_lab_coal_wide_samp, "train")
```

Do the same for the labels, y

```
# Now convert the labels (aka 'y','logits' or 'digits') for saving to file.
# Also change 10 to 0 for representing 0 AND 11 to 10 for representing blank.

CreateLabelTensor <- function(data, subset_name){

  subset <- data %>%
              filter(sampleset == subset_name) %>%
                #reset sequence to start from 1 for each set, thus matching images
                mutate(sequence = sequence-min(sequence)+1,
                       #represent 0 with a 0, instead of 10.
                       LabelA = ifelse(LabelA==10,0,LabelA),
                       LabelB = ifelse(LabelB==10,0,LabelB),
                       LabelC = ifelse(LabelC==10,0,LabelC),
                       LabelD = ifelse(LabelD==10,0,LabelD),
                       LabelE = ifelse(LabelE==10,0,LabelE)) %>%
                mutate(#represent blank with a 10, instead of 11
                       LabelA = ifelse(LabelA==11,10,LabelA),
                       LabelB = ifelse(LabelB==11,10,LabelB),
                       LabelC = ifelse(LabelC==11,10,LabelC),
                       LabelD = ifelse(LabelD==11,10,LabelD),
                       LabelE = ifelse(LabelE==11,10,LabelE)) %>%
                  select(sequence, DigitCount,LabelA,LabelB,LabelC,LabelD,LabelE) %>%
                    arrange(sequence)

  write_delim(subset, paste0("labeltensor_",subset_name,".txt"), delim = " ")
  print(paste0(subset_name, " labels: Done"))
  return(subset)
}

y_validate  <- CreateLabelTensor(bbox_dt_lab_coal_wide_samp, "validate")
y_test      <- CreateLabelTensor(bbox_dt_lab_coal_wide_samp, "test")
y_train     <- CreateLabelTensor(bbox_dt_lab_coal_wide_samp, "train")
```

## Building a TensorFlow Model

The next notebook will construct a tensorflow model to identify the digits in these images.