**LabelImg Download**

* To label images manually using YOLO labels format
* <http://tzutalin.github.io/labelImg/> (Prebuilt GUI version)
* <https://github.com/tzutalin/labelImg#labelimg>
* Edit predefined classes.txt

**Dataset**

* ~1200 manually labelled dataset
* ~500 from Kaggle dataset (<https://www.kaggle.com/imamdigmi/indonesian-plate-number>)
* 1472 Train Images, 201 Validation Images

**Prevent Colab from Timeout**

function ClickConnect(){

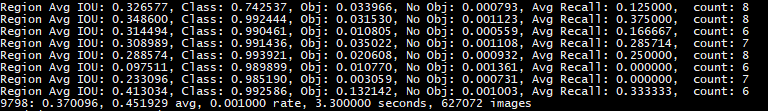
console.log("Clicked on connect button");

document.querySelector("colab-connect-button").click()

}

setInterval(ClickConnect,60000)

**YOLOV3**

* Need to have train + test folders in data folder
* In each folder contains IMGxxx.jpg and IMGxxx.txt
  + IMGxxx.txt contains:
    - Class\_label x y width height
    - Class\_label = encoded class label (0 - # classes-1)
    - X = x center of bounding box
    - Y = y center of bounding box
    - Width = width of bounding box
    - Height = height of bounding box
    - All Numbers (x, y, width, height) needs to be between 0-1
* git clone <https://github.com/pjreddie/darknet>
* Download YOLOv3 weights and cfg from <https://pjreddie.com/darknet/yolo/> (if needed)
* Change Makefile in Darknet folder to CPU = 1, CUDNN = 1, OPENCV = 1
* In yolov3.cfg:
  + Comment line 3 (batch=1), line 4 (subdivisions=1)
  + Replace classes = # classes (on each of the YOLO layers, just CNTRL+F “YOLO”)
  + Replace filters = (# classes + 5) \* 3 (on top of every yolo layers)
  + Max batches = 2000 \* # classes (# iterations)
  + Steps = 80%, 90% of max batches
* Create custom obj.names, obj.data
  + obj.names = name of classes (one per line in accordance with mapping)
  + obj.data
    - classes = # numclasses
    - train = data/train.txt
    - valid = data/test.txt
    - names = data/obj.names
    - backup = /mydrive/yolov3/backup (backup path)
* Download pretrained darknet weights: <https://pjreddie.com/media/files/darknet53.conv.74>
* Change checkpoint intervals: (detector.c line 138) Change to THIS!!!
  + To save every 100 iterations up to 10000 epochs
  + Original: if(i%10000==0 || (i < 1000 && i%100 == 0))
  + Changed: if(i%10000==0 || (i < 10000 && i%100 == 0))
* Creating symbolic links to avoid errors in google drive
  + !ln -s /content/gdrive/My\ Drive / /mydrive
    - Maps the original googledrive path to /mydrive
* YOLOv3 Output
  + 9798 = Indicates the number of iterations of the current training
  + 0.370096 = it is the overall Loss (loss)
  + 0.451929 avg = the average Loss, this value should be as low as possible. Once this value is less than 0.060730 avg may terminate training
  + 0.001000 rate = represents the current learning rate, is defined in the .cfg file
  + 3.300000 seconds = Indicates the current batch total time spent training
  + 627072 images = the last line of this value is the size of 9798 \* 64, expressed so far, the total amount involved in the training of the picture. (64 is batch size)
* To Calculate mAP of Model: <https://github.com/Cartucho/mAP>

(for validation of which weights are best on validation set 🡪 Done after training is complete)

* + Save model weights every 2000 iterations
  + Do prediction on validation set
  + Convert ground truth and prediction to mAP format for evaluation
    - mAP github format: classname, prob, left, top, right, bottom
    - YOLOv3 format: centerx, centery, width, height
    - Example:
      * Ground Truth: LP 2 10 173 238
      * Prediction: LP 0.9 1 9 154 200S
  + Place the outputs into input folder (ground-truth for labels, detection-results for prediction)
  + To run -> cd to directory of mAP github -> python main.py

**Text Recognition**

* Use Keras OCR for detection and Recognition
  + <https://github.com/faustomorales/keras-ocr>
* Weights for Detection:

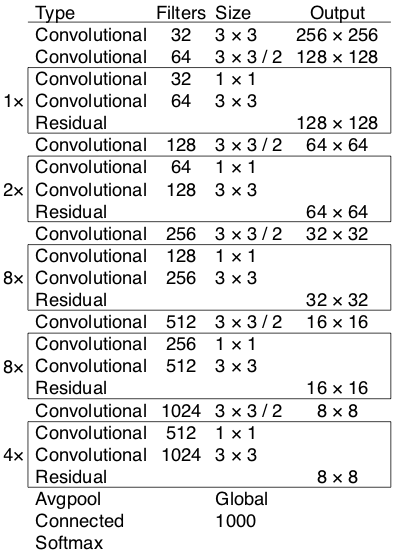
<https://www.mediafire.com/file/mepzf3sq7u7nve9/craft_mlt_25k.h5/file>

* Weights for Recognition: <https://www.mediafire.com/file/pkj2p29b1f6fpil/crnn_kurapan.h5/file>
* To combine: Output from YOLOv3 -> Apply text recognition

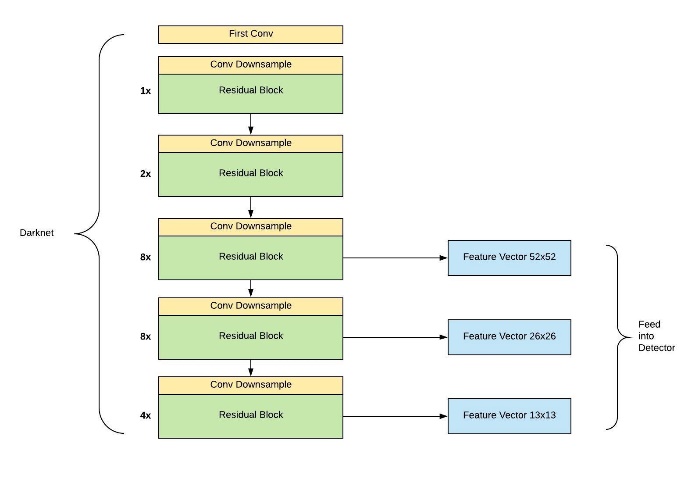
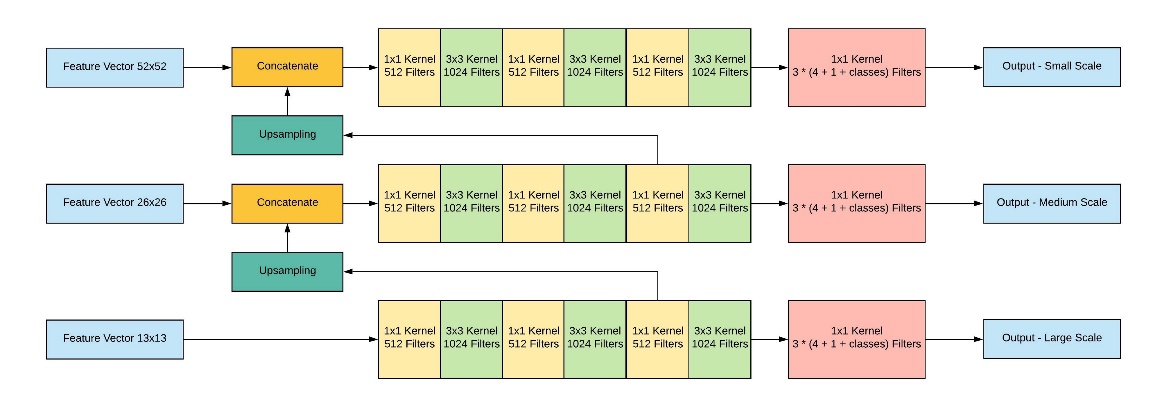
**Procedure**

* Gather dataset ~1200 hand labelled images, ~500 labelled dataset from Kaggle
* Convert labels to YOLOv3 format
  + Train and test folder in data folder
  + Labels.txt after every image in the same directory
  + Class label (encoded), centerx, centery, width, height
  + Train.txt = contains all path to all training images
  + Test.txt = contains all path to all test images
* Clone the YOLOv3 repository and edit configuration
* Train models for 2000 iterations -> Evaluate on hyperparameter -> pick best model
* Take output of YOLOv3 -> feed into KerasOCR model
* Display prediction -> DONE!

**Technical**

* YOLOv3
  + **Darknet-53 as base** 
    - 53 convolution layers each with Batch Norm and Leaky ReLU Activation

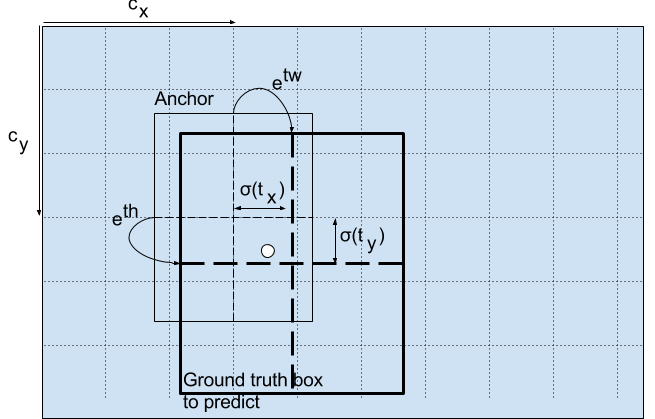
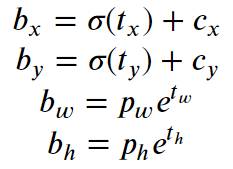
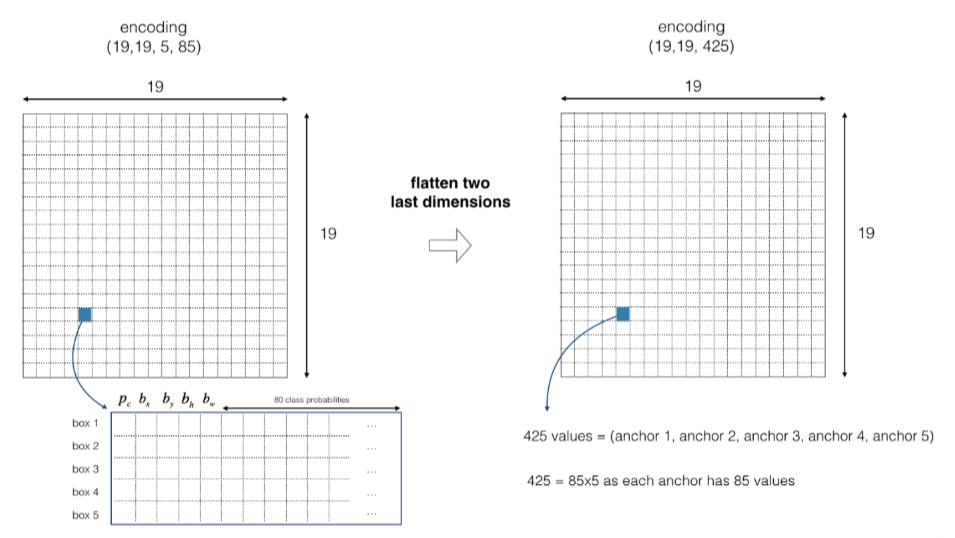
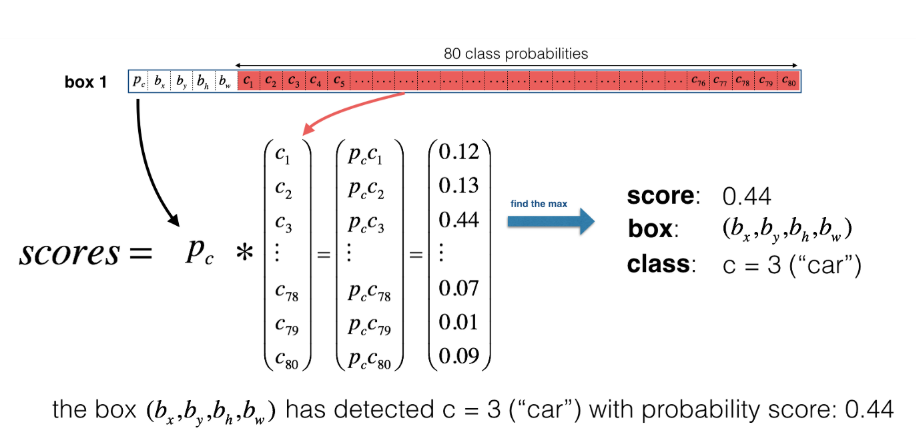
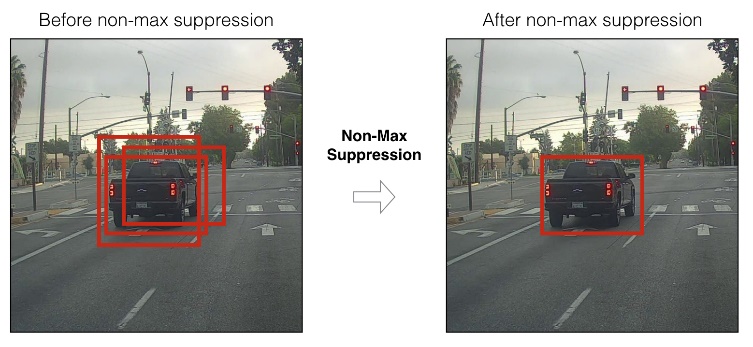
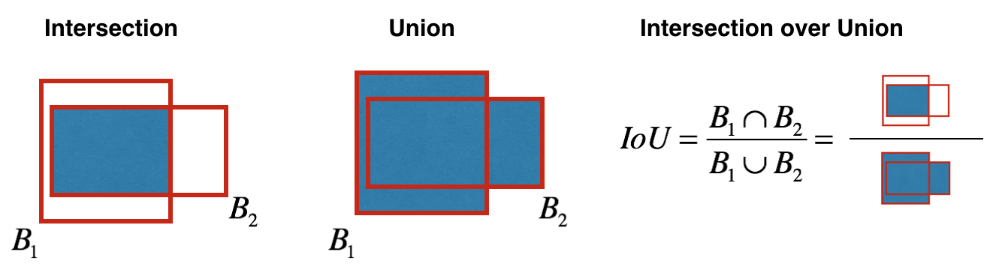
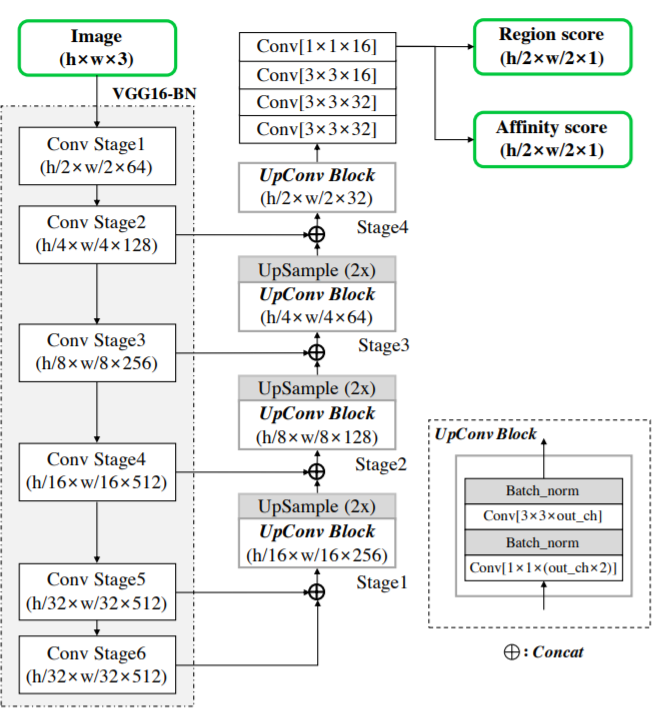
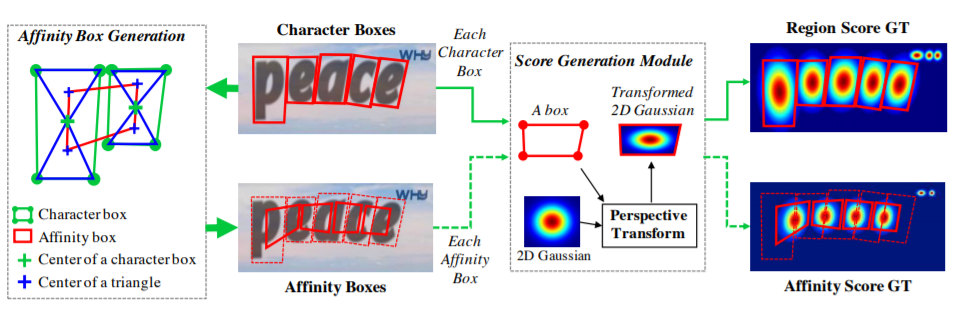
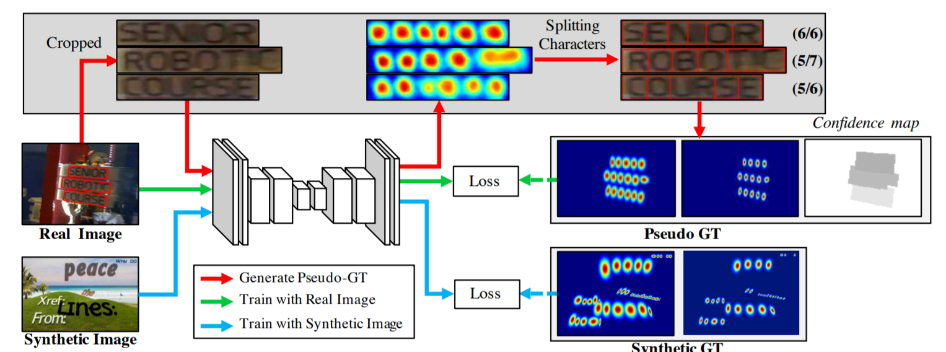
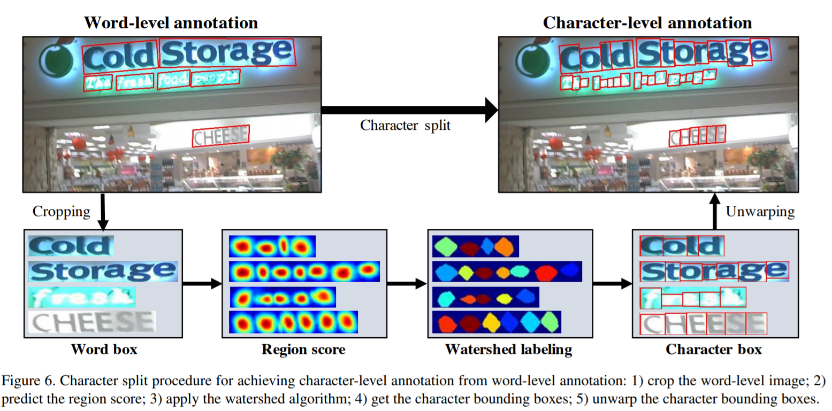
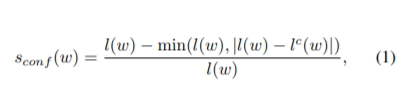
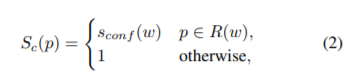
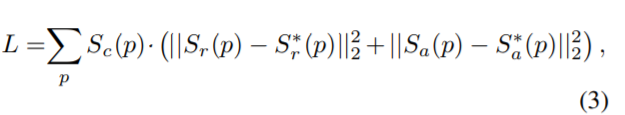
Darknet-53: Output numbers here don’t match. In YOLOv3 use input 416x416

* + - No pooling is used, and convolution of stride 2 is used
    - AVG pooling layer deleted -> append a detection head to Darknet53 (input: 416x416)
    - Features from last 3 blocks are extracted and used to feed into detector
    - **Detector:** used multiple 1x1 and 3x3 convolution
  + Invariant to input size, but in practice, use fixed input size (416x416) for parallelization by GPU
  + **Input:** (m, 416, 416, 3)
    - m = # of images
    - 416, 416 = input image size (fixed)
    - 3 = # channels (RGB)

(0,0)

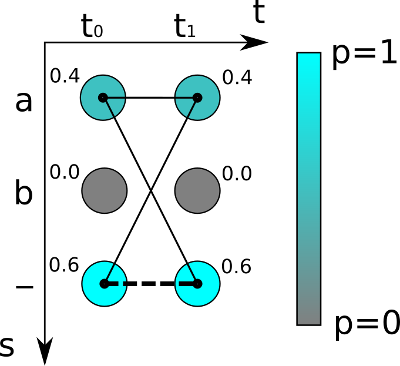
* + **Output:** (𝑝𝑐,𝑏𝑥,𝑏𝑦,𝑏ℎ,𝑏𝑤,𝑐)
    - pc = probability of class c (0 to 1)
    - bx, by = x,y center of bounding box
    - bh, bw = height and width of bounding box
    - c = # of classes (one hot encoded vector)
    - Example: (0.8, 0.25, 0.3, 0.5, 0.4, 0, 1, 0)

(1,1)

* + YOLOv3 predicts 3 anchor box per cell
  + **Anchor BOX:**
    - Anchor box is a prior box that could have different pre-defined aspect ratios.
    - **aspect ratios are determined before training by running K-means on the entire dataset**
    - 3 anchor box per cell -> 52x52x3, 26x26x3, 13x13x3 = 10647 bounding boxes
      * 1. The location offset against the anchor box: tx, ty, tw, th. This has 4 values.
      * 2. The objectness score to indicate if this box contains an object. This has 1 value.
      * 3. The class probabilities to tell us which class this box belongs to. This has num\_classes values.
  + **Log space output of network** (instead of bounding box coordinates due to unstable gradients)
    - This shows the conversion between tx, ty, tw, th (network output) with YOLOv3 labels
    - Prediction at 3 different scales
      * Stride of 32, 16, 8 respectively
      * Input of 416x 416 -> 13 x 13, 26 x 26, 52 x 52
      * In total: 416 x 416 input image
        + YOLO predicts ((52 x 52) + (26 x 26) + 13 x 13)) x 3 = 10647 bounding boxes
        + 3 = # of anchor boxes per cell
  + **Anchor box Example:**
    - Image = 608 x 608 -> stride 32 -> 19 x 19 (608/32)
    - **If the center/midpoint of an object falls into a grid cell, that grid cell is responsible for detecting that object.**
    - In this example: 5 anchor box per cell and 80 classes
    - 19 x 19 x 5 x 85 -> 19 x 19 x 425
      * 85 = 80 classes + pc + bx,by,bw,bh
      * Multiply to get output and take maximum
  + **Loss Function:**
    - Total loss = xy loss + wh loss + obj loss + no\_obj loss + class loss
    - tx, ty, tw, th as defined above using transformations
    - Lambda\_coord = 5 (focus on localization)
    - xy\_loss = Lambda\_Coord \* Sum(Mean\_Square\_Error((tx, ty), (tx’, ty’)) \* obj\_mask)
      * obj\_mask = 1 if there is an obj, 0 otherwise
    - wh\_loss = Lambda\_Coord \* Sum(Mean\_Square\_Error((tw, th), (tw’, th’)) \* obj\_mask)
    - obj\_loss = Sum(Binary\_Cross\_Entropy(obj, obj’) \* obj\_mask)
    - noobj\_loss = Lambda\_Noobj \* Sum(Binary\_Cross\_Entropy(obj, obj’) \* (1 — obj\_mask) \* ignore\_mask)
      * Lambda\_Noobj = 0.5 to make sure the network won’t be dominated by cells that don’t have objects.
      * The `ignore\_mask` is used to make sure we only penalize when the current box doesn’t have much overlap with the ground truth box.
    - Objectness indicates how likely is there an object in the current cell.
    - In the ground truth, objectness (obj) is always 1 for the cell that contains an object, and 0 for the cell that doesn’t contain any object.
    - class\_loss = Sum(Binary\_Cross\_Entropy(class, class’) \* obj\_mask)
  + **Output Processing**
    - Get rid of low confidence boxes (< threshold, usually 0.5)
    - **Non-max suppression**
      * Select the box that has the highest score.
      * Compute its overlap with all other boxes, and remove boxes that overlap it more than iou\_threshold.
      * Go back to step 1 and iterate until there is no more boxes with a lower score than the current selected box.
  + **References:**
    - <https://medium.com/analytics-vidhya/yolo-v3-theory-explained-33100f6d193>
    - <https://towardsdatascience.com/dive-really-deep-into-yolo-v3-a-beginners-guide-9e3d2666280e>
* **KerasOCR**
  + **Consists of 2 models Character Region Awareness for Text Detection (CRAFT) and CRNN for text recognition**
    - <https://github.com/clovaai/CRAFT-pytorch>
    - <https://github.com/kurapan/CRNN>
  + **CRAFT**
    - Use VGG16 as backbone
    - Left Image: network Architecture, Right Image: Label generation pipeline
    - Goal: localize each individual characters
    - The final output has two channels as score maps: the region score and the affinity score
    - How model is trained?
    - **Training Procedure:**
      * Word level images are cropped
      * Current training model predict region score
      * Watershed algorithm -> is used to split the character regions, which is used to make the character bounding boxes covering regions
      * Coordinates transformed back to original
    - Confidence score:
      * S\_conf(w) = confidence score for sample w
      * R(w) = bounding box region around word length
      * L(w) = word length of sample w
      * L^c(w) = estimated character length of bounding box
      * S\_c(p) = pixelwise confidence confidence map
      * P = pixel in region R(W)
      * Ignore boxes with S\_conf(w) < 0.5
    - Loss function:
      * S∗\_r(p) = pseudo-ground truth region score
      * S∗\_a(p) = pseudo-ground truth affinity map
      * S\_r(p) = predicted region score
      * S\_a(p) = predicted affinity score
      * When training with synthetic data, we can obtain the real ground truth, so S\_c(p) is set to 1, otherwise follow formula 2.
    - Paper link: <http://openaccess.thecvf.com/content_CVPR_2019/papers/Baek_Character_Region_Awareness_for_Text_Detection_CVPR_2019_paper.pdf>
  + **CRNN text recognition**
    - Take input of character bounding boxes from CRAFT model and detect
    - Use CNN to extract sequence of features and RNN to propagate information
    - CTC Loss function:
      * Connectionist Temporal Classification (CTC)
      * Why it is used?
        + we only tell the CTC loss function the text that occurs in the image. Therefore, we ignore both the position and width of the characters in the image.
        + no further processing of the recognized text is needed.
      * We only feed the output matrix of the NN and the corresponding ground-truth (GT) text to the CTC loss function.
      * Encoding the text
        + Duplicate characters are encoded with character `-`
        + Example:

“to” → “---ttttttooo”, or “-t-o-”, or “to”

“too” → “---ttttto-o”, or “-t-o-o-”, or “to-o”, but not “too”

* + - * The NN is trained to output encoded text.
      * Outputs a matrix containing a score for each character at each time-step.
      * Loss Calculation example:
        + 3 vocab (a, b, -), 2-time steps (t0, t1)
        + At each time step score sum to 1 (probability)
        + Further, you already know that the loss is calculated by summing up all scores of all possible alignments of the GT text, this way it does not matter where the text appears in the image.
        + Sum of scores:

Aa = 0.4 \* 0.4 = 0.16

A- = 0.4 \* 0.6 = 0.24

-a = 0.3 \* 0.6 = 0.24

Aa + a- + -a = 0.16 + 0.24 + 0.24 = 0.64

If ground truth text is “” THEN it corresponds to -- = 0.6 \* 0.6 = 0.36

Maximize probabilities -> Minimize negative of log loss

* + - * **Decoding**
        + Uses best path decoding (algo)

(Take maximum prob at each step)

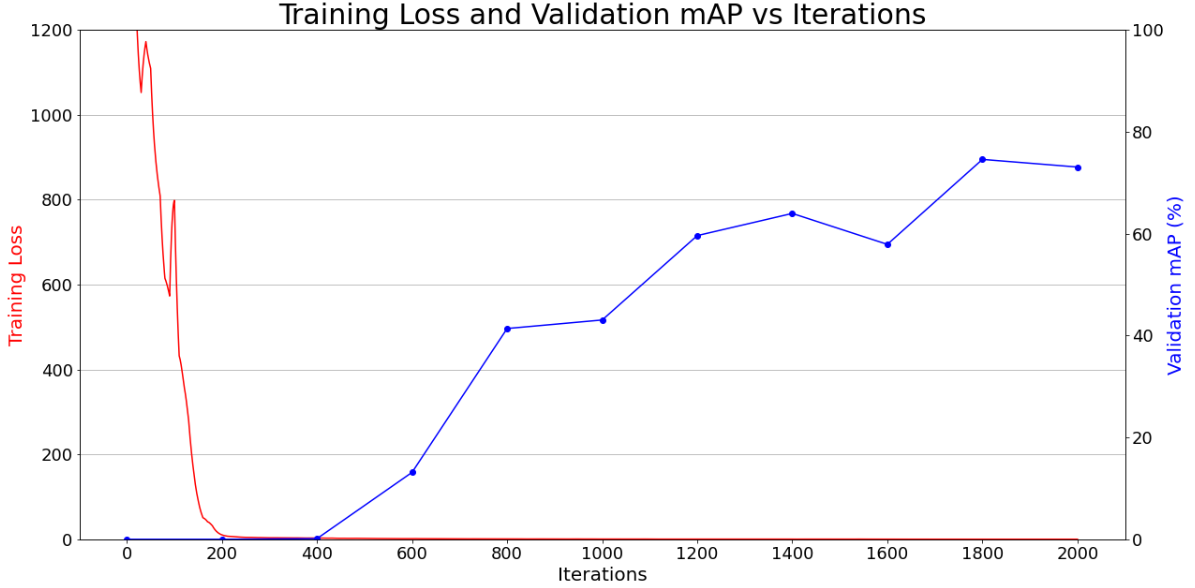
1. it calculates the best path by taking the most likely character per time-step.

2. it undoes the encoding by first removing duplicate characters and then removing all blanks from the path. What remains represents the recognized text.

* + - * + Can also do Beam Search
      * <https://towardsdatascience.com/intuitively-understanding-connectionist-temporal-classification-3797e43a86c>

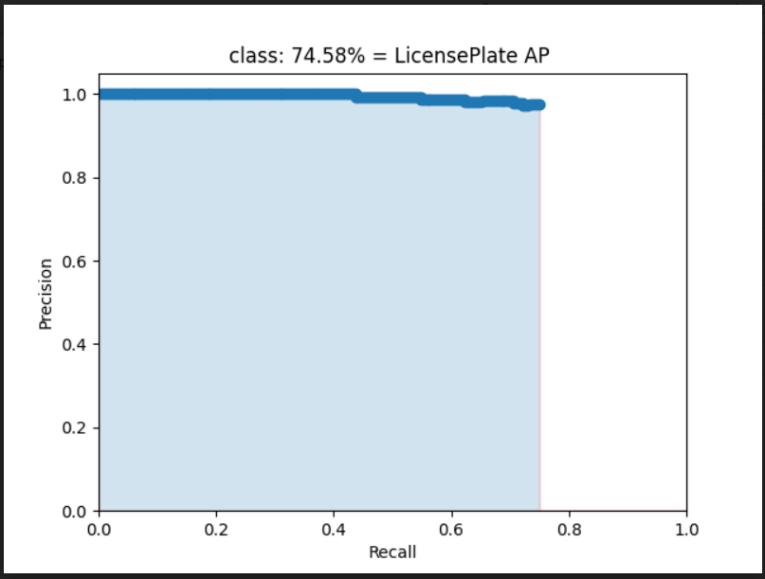
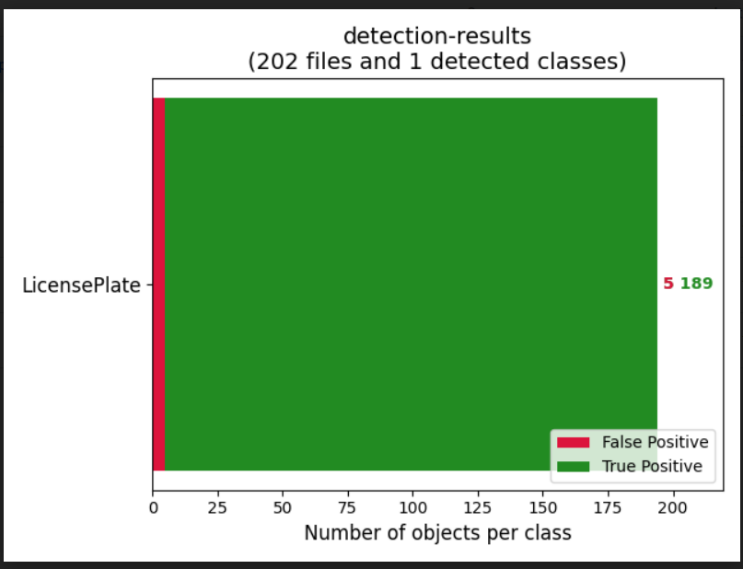


* + - Implementation of Model
      * Uses Bidirectional LSTM in 2 layers
      * Default parameters:
        + Width = 200, Height = 31, Channels = 1
        + Conv\_filter\_size = [64, 128, 256, 256, 512, 512, 512]
        + Lstm\_units = [128, 128]
        + Dropout rate = 0.25
    - See details in the github repo: <https://github.com/kurapan/CRNN>

**Training and Validation Results**

|  |  |
| --- | --- |
| **Iteration** | **Validation mAP** |
| 0 | 0% |
| 200 | 0% |
| 400 | 0.13% |
| 600 | 13.14% |
| 800 | 41.38% |
| 1000 | 43.05% |
| 1200 | 59.62% |
| 1400 | 63.97% |
| 1600 | 57.90% |
| 1800 | 74.58% |
| 2000 | 73.08% |

Best results used 1800 iterations, so this is the one chosen for implementation

Some more plots about the evaluation of 1800 iterations: (log-average miss rate = 0.27)

**Streamlit**

**To Run Streamlit App:** streamlit run xxx.py

**Docker**

* Create a new folder called docker to keep everything clean
* Cd into that directory (use quotes)
* Build Images (name is mystapp:latest): docker build -t mystapp:latest .
* Build Container (map to port 8080): docker run -it -p 8080:8080 mystapp:latest
* Access app with: <http://192.168.99.100:8080/>
* Dig into a file system of an already run container: docker exec -t -i mycontainer /bin/bash
* Install VIM into docker container:
  + apt-get update
  + apt-get install vim
* Stop container: docker stop containerID
* View all container: docker ps -a
* View all images: docker images
* Remove all containers: docker rm $(docker ps -aq)
* Remove all images: docker rmi $(docker images -q)

**Deploy Docker to GCP**

* <https://www.youtube.com/watch?v=03KgXhg-voY>
* Create app.yaml file
* Ensure that gcloud sdk is installed in local file system
* List of all projects: gcloud projects list
* Look at current project: gcloud config get-value project
* Change to project: gcloud config set project projectID
* Deploy: gcloud app deploy

**References:**

* <https://towardsdatascience.com/automatic-license-plate-detection-recognition-using-deep-learning-624def07eaaf>
* <https://towardsdatascience.com/i-built-a-diy-license-plate-reader-with-a-raspberry-pi-and-machine-learning-7e428d3c7401>