1. Introduction and building the model[¶](file:///D:\PYTHON\ssd_keras\train_ssd7.html#1.-Introduction-and-building-the-model)

The cell below sets a number of parameters that define the model architecture and then calls the function build\_model() to build the model. Read the comments and the documentation, but here are some further explanations for a few parameters:

* Set the height, width, and number of color channels to whatever you want the model to accept as image input. This does not have to be the actual size of your input images! However, if your input images have a different size than you define as the model input here, you must use the crop, resize and/or random\_crop features of the batch generator to convert your images to the model input size during training. If your dataset contains images of varying size, like the Pascal VOC datasets for example, use the random\_crop feature of the batch generator to cope with that (see the documentation).
* The number of classes includes the background class, i.e. if you have n positive classes in your dataset, set n\_classes = n + 1. Class ID 0 must always be reserved for the background class, i.e. your positive classes must have positive integers as IDs.
* The reason why the list of scaling factors has 5 elements even though there are only 4 predictor layers in this model is that the last scaling factor is used for the second aspect-ratio-1 box of the last predictor layer. See the documentation for details.
* Alternatively to passing an explicit list of scaling factors, you could also just define a mimimum and a maximum scale, in which case the other scaling factors would be linearly interpolated. If you pass both min/max scaling factors and an explicit list, the explicit list will be used.
* build\_model() and SSDBoxEncoder have two arguments for the anchor box aspect ratios: aspect\_ratios\_global and aspect\_ratios\_per\_layer. You can use either of the two. If you use aspect\_ratios\_global, then you just pass a list containing all aspect ratios for which you would like to create anchor boxes. Every aspect ratio you want to include must be listed once and only once. If you use aspect\_ratios\_per\_layer, then you pass a list containing lists of aspect ratios for each individual predictor layer. In the example below, the model has four predictor layers, so you would pass a list containing four lists.
* If two\_boxes\_for\_ar1 == True, then two boxes of different size will be created for aspect ratio 1 for each predictor layer. See the documentation for details.
* If limit\_boxes == True, then the generated anchor boxes will be limited so that they lie entirely within the image boundaries. This feature is called 'clip' in the original Caffe implementation. Even though it may seem counterintuitive, it is recommended **not** to clip the anchor boxes. According to Wei Liu, the model performs slightly better when the anchors are not clipped.
* The variances are scaling factors for the target coordinates. Leaving them at 1.0 for each of the four box coordinates means that they have no effect whatsoever. Decreasing them to below 1.0 **upscales** the gradient for the respective target box coordinate.
* The coords argument lets you choose what coordinate format the model should learn. If you choose the 'centroids' format, the targets will be converted to the (cx, cy, w, h) coordinate format used in the original implementation. If you choose the 'minmax' format, the targets will be converted to the coordinate format (xmin, xmax, ymin, ymax). The model, of course, will learn whatever the targets tell it to.
* normalize\_coords converts all absolute ground truth and anchor box coordinates to relative coordinates, i.e. to coordinates that lie within [0,1] relative to the image height and width. Whether you use absolute or relative coordinates has no effect on the training - the targets end up being the same in both cases. The main reason why the original implementation uses relative coordinates is because it makes coding some box operations more convenient. This defaults to False.

These paramters might be a bit much at first, but they allow you to configure many things easily.

The parameters set below are not only needed to build the model, but are also passed to the SSDBoxEncoder constructor in the subsequent cell, which is responsible for matching and encoding ground truth boxes and anchor boxes during training. In order to do that, it needs to know the anchor box specifications. It is for the same reason that build\_model() does not only return the model itself, but also predictor\_sizes, a list of the spatial sizes of the convolutional predictor layers - SSDBoxEncoder needs this information to know where the anchor boxes must be placed spatially.

The original Caffe implementation does pretty much everything inside a model layer: The ground truth boxes are matched and encoded inside [MultiBoxLossLayer](https://github.com/weiliu89/caffe/blob/ssd/src/caffe/layers/multibox_loss_layer.cpp), and box decoding, confidence thresholding and non-maximum suppression is performed in [DetectionOutputLayer](https://github.com/weiliu89/caffe/blob/ssd/src/caffe/layers/detection_output_layer.cpp). In contrast to that, in the current form of this implementation, ground truth box matching and encoding happens as part of the mini batch generation (i.e. outside of the model itself). To be specific, the generate() method of BatchGenerator calls the encode\_y() method of SSDBoxEncoder to encode the ground truth labels, and then yields the matched and encoded target tensor to be passed to the loss function. Similarly, the model here outputs the raw prediction tensor. The decoding, confidence thresholding, and non-maximum suppression (NMS) is then performed by decode\_y2(), i.e. also outside the model. It's (almost) the same process as in the original Caffe implmentation, it's just that the code is organized differently here, which likely has performance implications, but I haven't measured it yet. I might look into incorporating all processing steps inside the model itself, but for now it was just easier to take the non-learning-relevant steps outside of Keras/Tensorflow. This is one advantage of Caffe: It's more convenient to write complex custom layers in plain C++ than to grapple with the Keras/Tensorflow API.

Note:[¶](file:///D:\\PYTHON\\ssd_keras\\train_ssd7.html" \l "Note:)

The example setup below was used to train SSD7 on two street traffic datasets released by [Udacity](https://github.com/udacity/self-driving-car/tree/master/annotations) with around 20,000 images in total and 5 object classes (car, truck, pedestrian, bicyclist, traffic light), although the vast majority of the objects are cars. The original datasets have a constant image size of 1200x1920 RGB. I consolidated the two datasets, removed a few bad samples (although there are probably many more), and resized the images to 300x480 RGB, i.e. to one sixteenth of the original image size. In case you'd like to train a model on the same dataset, you can find the consolidated and resized dataset I used [here](https://drive.google.com/file/d/0B0WbA4IemlxlT1IzQ0U1S2xHYVU/view?usp=sharing).

2. Set up the training

The cell below sets up everything necessary to train the model. The only things you have to set are the correct file paths to the images and labels in your dataset, and in case your labels do not come in a CSV file, you might have to switch from the CSV parser to the XML parser or you might have to write a new parser method in the BatchGenerator class that can handle whatever the format of your labels is. The README of this project provides an overview of the design of the batch generator class, which should help you in case you need to write a new parser or adapt one of the existing parsers to your needs.

For everything in this cell that does not concern loading your data: You don't have to change anything (but you can change everything of course).

Set the batch size to whatever value you like (and one that makes the model fit inside your GPU memory), it's not the most important hyperparameter - 32 works well, but so do most other batch sizes.

I'm using an Adam optimizer with the standard initial learning rate of 0.001 and a small decay, nothing special.

SSDLoss is a custom Keras loss function that implements the multi-task log loss for classification and smooth L1 loss for localization. neg\_pos\_ratio and alpha are set as in the paper and n\_neg\_min is a rather unimportant optional parameter to make sure that a certain number of negative boxes always enters the loss function even if there are very few or no positive boxes in a batch, which should never happen anyway.

The ssd\_box\_encoder object, which, as explained above, knows how to match and encode the ground truth labels into the format that the model needs, is passed to the batch generator, which during training loads the next batch of images and labels, optionally performs data augmentation, and encodes the ground truth labels.

There are two parameters in the SSDBoxEncoder that you should note: pos\_iou\_threshold and neg\_iou\_threshold. The former determines the minimum Jaccard overlap between a ground truth box and an anchor box for a match and is set to 0.5, the value stated in the paper. The latter, neg\_iou\_threshold, is not in the paper, but it is useful to improve the learning process. It determines the maximum allowed Jaccard overlap between an anchor box and any ground truth box in order for that anchor box to be considered a negative box. This is useful because you want a clear margin between negative and positive boxes. An anchor box that almost contains an object should not be forced to learn to predict a negative box in such a case. 0.2 is a reasonable value that is used by various other object detection models.

3. Run the training

Now that everything is set up, we're ready to start training. Set the number of epochs and the model name, the weights name in ModelCheckpoint and the filepaths to wherever you'd like to save the model. There isn't much more to say here, just execute the cell. If you get "out of memory" errors during training, reduce the batch size.

Training currently only monitors the validation loss, not the mAP.

4. Make predictions

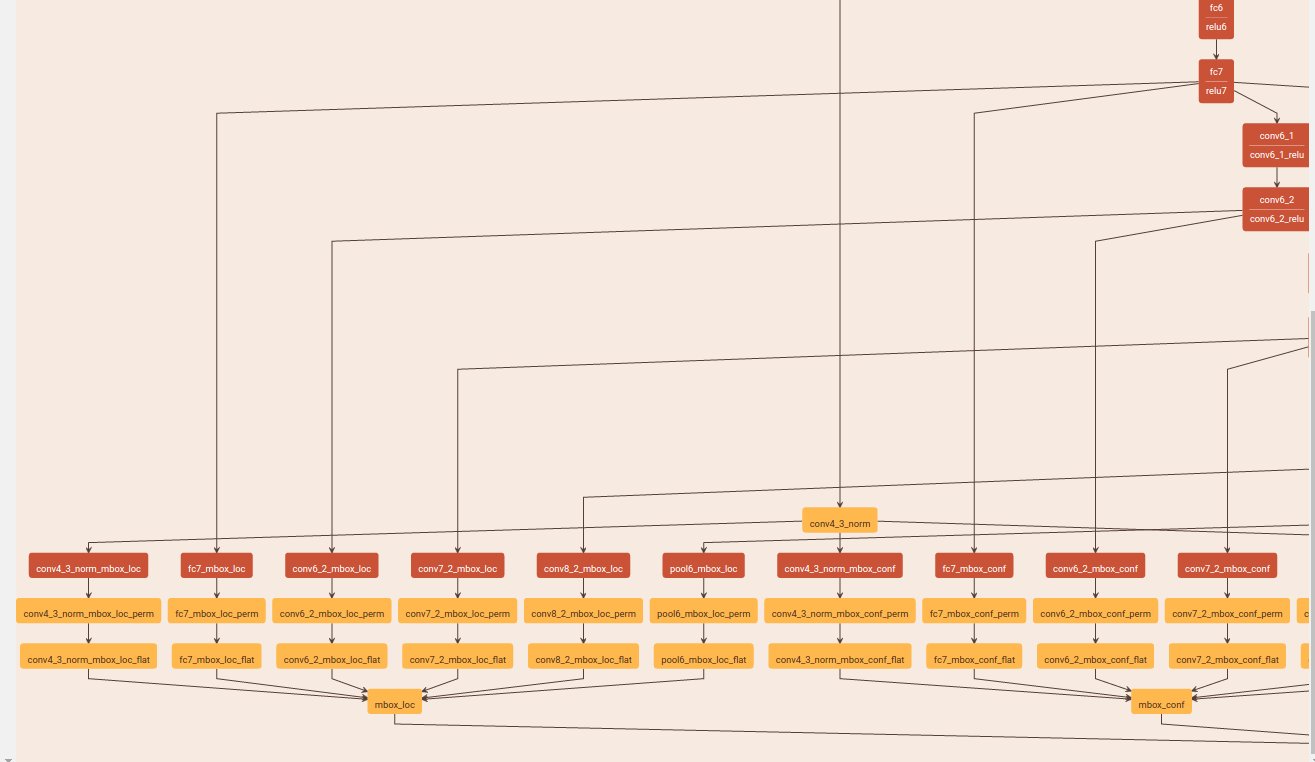
Now let's make some predictions on the validation dataset with the trained model. We'll use the validation generator which we've already set up above. If you did not use a validation dataset, change "val\_dataset" to "train\_dataset" below (or whatever you called the BatchGenerator instance you used for training above). Feel free to change the batch size.

Now let's decode the raw prediction y\_pred. The function decode\_y2() converts the box coordinates from offsets back to absolute coordinates, keeps only the positive predictions (i.e. throws away all boxes for which the highest confidence is for class 0), applies a confidence threshold to all positive predictions, and applies non-maximum suppression to the remaining predictions, in this order. In case you would like to omit the NMS step, set iou\_threshold = None.

You could also use decode\_y(), which follows the prodecure outlined in the paper, to decode the raw predictions, but decode\_y2() is more efficient and I found it yields better results. Check out the documentation for details on how the two decoders differ.

Finally, let's draw the predicted boxes onto the image in blue to visualize the result. Each predicted box says its confidence next to the category name. The ground truth boxes are also drawn onto the image in green for comparison.

疑问1：



每个预测分支中3x3卷积的作用，我用train模型的可视化图来说明：例如从fc7（实际是一个卷积层）的feature map引出的loc分支，fc7\_mbox\_loc是一个卷积层，有6x4=24个3x3卷积核。其中，6是anchor的种类，4是回归预测的4个坐标（的offset），即每个cell都分别给6种anchor box预测4个坐标。得到24个feature map后再flatten拉长，最后把所有层的特征都拼接起来，求置信度conf的分支也类似，最后mbox\_loc和mbox\_conf送到整体损失层mbox\_loss。