WildFSL

Release 1.0.0

Arren Antioquia, Carl Dela Cruz, Jericho Dizon, Russel Campol

Apr 01, 2024

CHAPTER

ONE

SFDET-PYTORCH-WILDFSL

1.1 backbone package

1.1.1 Submodules

1.1.2 backbone.vgg module

class backbone.vgg.**VGG**(*config*, *in_channels*, *batch_norm=False*)

Bases: object

VGG classification architecture

get_layers()

Forms the layers of the model based on self.config

Returns:
list – contains all layers of VGG model based on self.config

1.1.3 Module contents

1.2 data package

1.2.1 Submodules

1.2.2 data.augmentations module

class data.augmentations.**Augmentations**(*size*, *mean*)

Bases: object

This class applies different augmentation techniques to the image. This is used for training the model.

class data.augmentations.**BaseTransform**(*size*, *mean*)

Bases: object

This class applies different base transformation techniques to the image. This includes resizing the image and subtracting the mean from the image pixels. This is used for testing the model.

class data.augmentations.**Compose**(*transforms*)

Bases: object

This class applies a list of transformation to an image.

class data.augmentations.**ConvertColor**(*current='BGR'*, *transform='HSV'*)

Bases: object

This class converts the image to another color space. This class supports the conversion from BGT to HSV color space and vice-versa.

class data.augmentations.**ConvertToFloat**

Bases: object

This class casts an np.ndarray to floating-point data type.

class data.augmentations.**Expand**(*mean*)

Bases: object

This class produces an image

class data.augmentations.**PhotometricDistort**

Bases: object

This class applies different transformation to the image. This includes adjustment of contrast, saturation, hue, and brightness, and the switching of channels.

class data.augmentations.**RandomBrightness**(*delta=32.0*)

Bases: object

This class adjusts the brightness of the image. This adds a random constant to the pixel values of the image.

class data.augmentations.**RandomContrast**(*lower=0.5*, *upper=1.5*)

Bases: object

This class adjusts the contrast of the image. This multiplies a random constant to the pixel values of the image.

class data.augmentations.**RandomHue**(*delta=18.0*)

Bases: object

This class adjusts the hue of the image. This expects an image in HSV color space, where the 1st channel (hue channel) should have values between 0.0 to 360.0

class data.augmentations.**RandomLightingNoise**

Bases: object

This class randomly swaps the channels of the image to create a lighting noise effect. This class calls the class SwapChannels.

class data.augmentations.**RandomMirror**

Bases: object

This class randomly flips the image horizontally. This also flips the coordinate of the bounding boxes of the objects.

class data.augmentations.**RandomSampleCrop**

Bases: object

class data.augmentations.**RandomSaturation**(*lower=0.5*, *upper=1.5*)

Bases: object

This class adjusts the saturation of the image. This expects an image in HSV color space, where the 2nd channel (saturation channel) should have values between 0.0 to 1.0

class data.augmentations.**Resize**(*size=300*)

Bases: object

This class resizes the image to output size

class data.augmentations.**SubtractMeans**(*mean*)

Bases: object

This class subtracts the mean from the pixel values of the image

class data.augmentations.**SwapChannels**(*swaps*)

Bases: object

This class swaps the channels of the image

class data.augmentations.**ToAbsoluteCoords**

Bases: object

This class converts the coordinates of bounding boxes to non-scaled values

class data.augmentations.**ToCV2Image**

Bases: object

This class converts the torch.Tensor representation of an image to np.ndarray. The channels of the image are also converted from RGB to BGR

class data.augmentations.**ToPercentCoords**

Bases: object

This class converts the coordinates of bounding boxes to scaled values with respect to the width and the height of the image

class data.augmentations.**ToTensor**

Bases: object

This class converts the np.ndarray representation of an image to torch.Tensor. The channels of the image are also converted from BGR to RGB

data.augmentations.**intersect**(*box_a*, *box_b*)

Compute the intersection areas between two sets of bounding boxes.

Arguments:

box_a {numpy.ndarray} – An array of shape (N, 4) representing N bounding boxes,
where each bounding box is represented as [x1, y1, x2, y2].

box_b {numpy.ndarray} – An array representing a single bounding box as [x1, y1, x2, y2].

Returns:

numpy.ndarray – An array containing the intersection areas between each bounding box in box_a
and the bounding box in box_b.

data.augmentations.**jaccard_numpy**(*box_a*, *box_b*)

Compute the Jaccard overlap (IoU) between two sets of bounding boxes.

Arguments:

box_a {numpy.ndarray} – An array of shape (N, 4) representing N bounding boxes,
where each bounding box is represented as [x1, y1, x2, y2].

box_b {numpy.ndarray} – An array representing a single bounding box as [x1, y1, x2, y2].

Returns:

numpy.ndarray – An array containing the Jaccard overlap (IoU) between each bounding box in box_a
and the bounding box in box_b.

1.2.3 data.data_loader module

data.data_loader.**detection_collate**(*batch*)

Collate function for combining a batch of samples into a batch of images and targets.

Args:

batch {list} – A list of samples, where each sample is a tuple (image, target).

- image (numpy.ndarray or torch.Tensor) – The input image.
- target (numpy.ndarray or torch.Tensor) – The target associated with the image.

Returns:

tuple – A tuple containing:

- torch.Tensor: A tensor containing a batch of images.
- list of torch.Tensor: A list containing tensors representing the targets associated with each image in the batch.

data.data_loader.**get_loader**(*config*)

Gets the DataLoader based on the provided configuration

Arguments: config – parsed config from main

Returns: data_loader – returns the DataLoader object depending on the configuration

1.2.4 data.wildfsl module

class data.wildfsl.**VOCAnnotationTransform**(*keep_difficult=False*)

Bases: object

This class converts the data from Annotation XML files to a list of [xmin, ymin, xmax, ymax, class]

class data.wildfsl.**WildFSL**(*data_path*, *new_size*, *mode*, *image_transform*, *target_transform=<data.wildfsl.VOCAnnotationTransform object>*, *keep_difficult=False*)

Bases: Dataset

WildFSL dataset

Extends:
Dataset

pull_annotation(*index*)

Gets the annotation of the image found in position index of the list of images in the dataset. The coordinates of the annotation is not scaled by the height and width of the image

Arguments:
index {int} – index of the image in the list of images in the dataset

Returns:
string, list – id of the image, list of bounding boxes of objects in the image formatted as [xmin, ymin, xmax, ymax, class]

pull_image(*index*)

Gets the image found in position index of the list of images in the dataset

Arguments:
index {int} – index of the image in the list of images in the dataset

Returns:
np.ndarray – image

pull_item(*index*)

Gets the image found in position index of the list of images in the dataset together with its corresponding annotation, its height, and its width

Arguments:
index {int} – index of the image in the list of images in the dataset

Returns:
torch.Tensor, np.ndarray, int, int – tensor representation of the image, list of bounding boxes of objects in the image formatted as [xmin, ymin, xmax, ymax, class], height, width

pull_tensor(*index*)

Gets a tensor of the image found in position index of the list of images in the dataset

Arguments:
index {int} – index of the image in the list of images in the dataset

Returns:
torch.Tensor – tensor representation of the image

data.wildfsl.**do_python_eval**(*results_path*, *dataset*, *output_txt*, *mode*, *iou_threshold*, *use_07_metric*)

Perform evaluation of object detection results using PASCAL VOC criteria.

Args:
results_path {str} – path to the directory containing detection results for all classes dataset (WildFSL) – instance of the WildFSL dataset class containing dataset information output_txt {str} – path to the output text file for logging evaluation results mode {str} – evaluation mode, either 'train' or 'test' iou_threshold (float) – Intersection over Union (IoU) threshold for considering a detection as correct use_07_metric (bool) – whether to use the 11-point interpolation method (VOC 07 metric) for computing AP

Returns:
tuple: A tuple containing APs for all classes and the mean Average Precision (mAP)

data.wildfsl.**parse_annotation**(*file_name*)

Parse a PASCAL VOC xml file

data.wildfsl.**save_results**(*all_boxes*, *dataset*, *results_path*, *output_txt*)

Saves results for each class

Arguments: dataset {dataset} – dataset object results_path {string} – path to save the results to output_txt {txt file} – results in a txt file

data.wildfsl.**voc_ap**(*recall*, *precision*, *use_07_metric=True*)

ap = voc_ap(rec, prec, [use_07_metric]) Compute VOC AP given precision and recall. If use_07_metric is true, uses the VOC 07 11 point method (default:True).

data.wildfsl.**voc_eval**(*detection_path*, *path*, *annotation_path*, *list_path*, *class_name*, *cache_dir*, *output_txt*, *iou_threshold=0.5*, *use_07_metric=True*)

Evaluate object detection performance using PASCAL VOC criteria.

Args:

detection_path {str} – path to the file containing detection results for a specific class path {str} – path to the dataset annotation_path {str} – path for XML annotations list_path {str} – path to the text file containing the list of image filenames to evaluate class_name {str} – name of the class for which to evaluate detection performance cache_dir {str} – directory to store cached annotations for faster processing output_txt {str} – path to the output text file for logging evaluation results iou_threshold {float, optional} – Intersection over Union (IoU) threshold for considering a detection as correct. Default is 0.5 use_07_metric {bool, optional} – whether to use the 11-point interpolation method (VOC 07 metric) for computing AP. Default is True

Returns:

tuple – A tuple containing recall, precision, and Average Precision (AP) for the specified class

1.2.5 Module contents

1.3 layers package

1.3.1 Submodules

1.3.2 layers.anchor_box module

`class layers.anchor_box.AnchorBox(new_size, config, scale_initial, scale_min, scale_max)`

Bases: `object`

Anchor box

`get_boxes()`

Computes location of anchor boxes in center-offset form for each feature map

Returns:

tensor – anchor boxes

`get_scales(scale_initial=0.1, scale_min=0.2, scale_max=1.05)`

Computes the scales used in the computation of anchor boxes

Keyword Arguments:

scale_min {float} – [description] (default: {0.2}) scale_max {float} – [description] (default: {1.05})

Returns:

list – contains the scales

1.3.3 layers.block module

`class layers.block.BasicConv(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, relu=True, bn=False, bias=True, up_size=0)`

Bases: `Module`

`forward(x)`

Forward pass of the BasicConv module.

Arguments:

x {torch.Tensor} – Input tensor.

Returns:

torch.Tensor – Output tensor after convolution, batch normalization, ReLU activation, and upsampling.

class layers.block.**Conv**(*in_channels*, *out_channels*, *kernel_size=1*, *padding=0*, *stride=1*, *dilation=1*, *groups=1*, *activation=True*)

Bases: Module

Conv module performs a convolution operation followed by batch normalization and LeakyReLU activation.

Arguments:

in_channels {int} – Number of input channels. out_channels {int} – Number of output channels. kernel_size {int or tuple, optional} – Size of the convolution kernel. Default is 1. padding {int or tuple, optional} – Padding of the convolution. Default is 0. stride {int or tuple, optional} – Stride of the convolution. Default is 1. dilation {int or tuple, optional} – Dilation rate of the convolution. Default is 1. groups {int, optional} – Number of groups for grouped convolution. Default is 1. activation {bool, optional} – Whether to apply LeakyReLU activation. Default is True.

forward(*x*)

Forward pass of the Conv module.

Arguments:

x {torch.Tensor} – Input tensor.

Returns:

torch.Tensor – Output tensor after convolution, batch normalization, and LeakyReLU activation.

class layers.block.**ReOrgLayer**(*stride*)

Bases: Module

ReOrgLayer module performs reorganization of feature maps by rearranging spatial positions.

Arguments:

stride {int} – Stride for reorganization.

forward(*x*)

Forward pass of the ReOrgLayer module.

Arguments:

x {torch.Tensor} – Input tensor.

Returns:

torch.Tensor – Output tensor after reorganization of feature maps.

1.3.4 layers.detection module

class layers.detection.**Detect**(**args*, ***kwargs*)

Bases: Function

static forward(*ctx*, *class_count*, *conf_data*, *loc_data*, *anchors*, *variance=[0.1, 0.2]*)

Forward pass of the Detect function.

Arguments:

ctx {torch.autograd.function.Context} – A context object to save information for the backward pass. class_count {int} – Number of classes. conf_data {torch.Tensor} – Confidence data. loc_data {torch.Tensor} – Localization data. anchors {torch.Tensor} – Anchor boxes. variance {list, optional} – List of variance values. Default is [0.1, 0.2].

Returns:

tuple: A tuple containing:

- torch.Tensor – Predicted bounding boxes.

- torch.Tensor – Confidence scores for each class.

1.3.5 layers.l2_norm module

class layers.l2_norm.**L2Norm**(*channels*, *scale*)

Bases: Module

L2Norm module performs L2 normalization on input feature maps.

Arguments:
channels {int} – Number of channels in the input feature maps. scale {float} – Scaling factor for the L2 normalization.

forward(*x*)

Forward pass of the L2Norm module.

Arguments:
x {torch.Tensor} – Input feature maps.

Returns:
torch.Tensor – Output feature maps after L2 normalization.

reset_parameters()

Reset the weight parameters with constant values.

1.3.6 layers.rainbow_module module

class layers.rainbow_module.**RainbowModule300**

Bases: Module

Rainbow Module 300

This module is designed to handle feature maps at multiple scales and perform operations such as convolution, batch normalization, and concatenation to generate fused feature maps.

Attributes:
layers {nn.ModuleList} – List of convolutional layers for each scale of feature map.

Methods:
init_weights – Initialize the weights of convolutional and batch normalization layers. forward – Perform the forward pass through the module.

forward(*x*)

Forward pass of the RainbowModule300 module.

Arguments:
x {list} – List of input feature maps.

Returns:
list – List of concatenated feature maps at different scales.

init_weights()

Initialize weights for convolutional and batch normalization layers.

class layers.rainbow_module.**RainbowModule512**

Bases: Module

RainbowModule512 – Module for multi-scale feature fusion with input feature maps of size 64x64.

This module consists of a series of convolutional and pooling layers for multi-scale feature fusion.

Attributes:
layers {nn.ModuleList} – List of modules for each scale of feature map.

Methods:
init_weights() – Initialize the weights of convolutional and batch normalization layers. forward(x) – Perform the forward pass through the module.

forward(*x*)

Perform the forward pass through the module.

Arguments:
x {list} – List of input feature maps at different scales.

Returns:
list – List of concatenated feature maps at different scales.

init_weights()

Initialize the weights of convolutional and batch normalization layers.

This method initializes the weights of convolutional layers using Xavier initialization and batch normalization layers with constant weights and biases.

1.3.7 layers.scale_transfer_module module

class layers.scale_transfer_module.**ScaleTransferModule**(*new_size*)

Bases: Module

Scale Transfer Module

This module is designed to transfer feature maps between different scales by performing operations such as average pooling and pixel shuffling.

Attributes:
new_size {int} – The target size of the feature maps after scaling. module_list {nn.ModuleList} – List of modules for transferring feature

maps between different scales.

Methods:
get_modules: Generate a list of modules based on the target size. forward: Perform the forward pass through the module.

forward(*x*)

Perform the forward pass through the module.

Arguments:
x {list} – List of input feature maps.

Returns:
list – List of output feature maps after scaling.

`get_modules()`

Generate a list of modules based on the target size.

Returns:

nn.ModuleList – List of modules for transferring feature maps
between different scales.

1.3.8 Module contents

1.4 loss package

1.4.1 Submodules

1.4.2 loss.loss module

`loss.loss.get_loss(config)`

Get the loss function based on the provided configuration.

Arguments:
config {dict} – A dictionary containing configuration parameters.

Returns:

loss – The selected loss function instance.

- For 'multibox' configuration, returns an instance of MultiBoxLoss.
- If no valid configuration is provided, returns None.

1.4.3 loss.multibox_loss module

`class loss.multibox_loss.MultiBoxLoss(class_count, iou_threshold, pos_neg_ratio, use_gpu)`

Bases: `Module`

MultiBoxLoss implements the loss function for object detection using SSD.

Arguments:
class_count {int} – The number of classes. iou_threshold {float} – The threshold for matching ground truth boxes with anchor boxes. pos_neg_ratio {float} – The ratio of negative to positive examples for hard negative mining. use_gpu {bool} – Flag indicating whether to use GPU.

Attributes:
variance {list} – The variance used for encoding bounding boxes.

Methods:
log_sum_exp: Computes the log sum exp function. forward: Computes the multi-box loss.

`forward(class_preds, class_targets, loc_preds, loc_targets, anchors)`

Computes the multi-box loss.

Arguments:
class_preds {torch.Tensor} – Predicted class scores. class_targets {torch.Tensor} – Ground truth class labels. loc_preds {torch.Tensor} – Predicted bounding box offsets. loc_targets {torch.Tensor} – Ground truth bounding box offsets. anchors {torch.Tensor} – Anchor boxes.

Returns:

tuple: A tuple containing the following:

- class_loss {torch.Tensor} – The classification loss.
- loc_loss {torch.Tensor} – The localization loss.
- loss {torch.Tensor} – The total loss.

`log_sum_exp`(*x*)

Computes the log sum exp function.

Arguments:
x {torch.Tensor} – Input tensor.

Returns:
torch.Tensor – Result of the log sum exp function.

1.4.4 Module contents

1.5 main module

`main.main`(*version*, *config*, *output_txt*)

`main.save_config`(*path*, *version*, *config*)

saves the configuration of the experiment

Arguments:
path {str} – save path version {str} – version of the model based on the time config {dict} – contains argument and its value

`main.string_to_boolean`(*v*)

Converts string to boolean

Arguments:
v {string} – string representation of a boolean values; must be true or false

Returns:
boolean – boolean true or false

`main.zip_directory`(*path*, *zip_file*)

Stores all py and cfg project files inside a zip file

Arguments:
path {string} – current path zip_file {zipfile.ZipFile} – zip file to contain the project files

1.6 models package

1.6.1 Submodules

1.6.2 models.model module

`models.model.get_model`(*config*, *anchors*, *output_txt*)

returns the model

1.6.3 models.sfdet_densenet module

class models.sfdet_densenet.**SFDetDenseNet**(*mode*, *base*, *fusion_module*, *pyramid_module*, *head*, *anchors*, *class_count*)

Bases: Module

SFDet DenseNet architecture for object detection.

Arguments:
mode {str} – Operating mode, either 'train' or 'test'. base {torch.nn.Module} – Base DenseNet model. fusion_module {list} – List of fusion modules. pyramid_module {list} – List of pyramid modules. head {tuple} – Tuple of lists containing class head and location head layers. anchors {torch.Tensor} – Anchors for bounding box predictions. class_count {int} – Number of object classes.

forward(*x*)

Forward pass of the SFDet DenseNet model.

Arguments:
x {torch.Tensor} – Input tensor.

Returns:
torch.Tensor – Predictions.

init_weights(*model_save_path*, *base_network*)

Initialize model weights.

Arguments:
model_save_path {str} – Path to save the model. base_network {str} – Base network for the model.

models.sfdet_densenet.**build_SFDetDenseNet**(*mode*, *new_size*, *densenet_model*, *anchors*, *class_count*)

Build the SFDet DenseNet model.

Arguments:
mode {str} – Operating mode, either 'train' or 'test'. new_size {int} – Size of the input images. densenet_model {str} – Name of the DenseNet model to use. anchors {torch.Tensor} – Anchors for bounding box predictions. class_count {int} – Number of object classes.

Returns:
SFDetDenseNet – SFDet DenseNet model.

models.sfdet_densenet.**get_fusion_module**(*config*, *in_channels*)

Get fusion modules based on configuration.

Arguments:
config {dict} – Configuration for fusion modules. in_channels {dict} – Input channels for fusion modules.

Returns:
list: List of fusion modules.

models.sfdet_densenet.**get_pyramid_module**(*config*)

Get pyramid modules based on configuration.

Arguments:
config {dict} – Configuration for pyramid modules.

Returns:
list – List of pyramid modules.

models.sfdet_densenet.**multibox**(*config*, *class_count*)

Create multibox head layers.

Arguments:
config {dict} – Configuration for multibox layers. class_count {int} – Number of object classes.

Returns:
tuple – Tuple of lists containing class head and location head layers.

1.6.4 models.sfdet_resnet module

class models.sfdet_resnet.**SFDetResNet**(*mode*, *base*, *fusion_module*, *pyramid_module*, *head*, *anchors*, *class_count*)

Bases: Module

SFDet ResNet architecture for object detection.

Arguments:
mode {str} – Operating mode, either 'train' or 'test'. base {torch.nn.Module} – Base ResNet model. fusion_module {list} – List of fusion modules. pyramid_module {list} – List of pyramid modules. head {tuple} – Tuple of lists containing class head and location head layers. anchors {torch.Tensor} – Anchors for bounding box predictions. class_count {int} – Number of object classes.

forward(*x*)

Forward pass of the SFDet ResNet model.

Arguments:
x {torch.Tensor} – Input tensor.

Returns:
torch.Tensor – Predictions.

init_weights(*model_save_path*, *base_network*)

Initialize model weights.

Arguments:
model_save_path {str} – Path to save the model. base_network {str} – Base network for the model.

models.sfdet_resnet.**build_SFDetResNet**(*mode*, *new_size*, *resnet_model*, *anchors*, *class_count*, *model_save_path*, *pretrained_model*, *output_txt*)

Build the SFDet ResNet model.

Arguments:
mode {str} – Operating mode, either 'train' or 'test'. new_size {int} – Size of the input images. resnet_model {str} – Name of the ResNet model to use. anchors {torch} –ensor): Anchors for bounding box predictions. class_count {int} – Number of object classes. model_save_path {str} – Path to save the model. pretrained_model {str} – Path to pretrained model. output_txt {str} – Path to output text.

Returns:
SFDetResNet – SFDet ResNet model.

models.sfdet_resnet.**get_fusion_module**(*config*, *in_channels*)

Get fusion modules based on configuration.

Arguments:
config {dict} – Configuration for fusion modules. in_channels {dict} – Input channels for fusion modules.

Returns:
list – List of fusion modules.

models.sfdet_resnet.**get_pyramid_module**(*config*)

Get pyramid modules based on configuration.

Arguments:
config {dict} – Configuration for pyramid modules.

Returns:
list – List of pyramid modules.

models.sfdet_resnet.**multibox**(*config*, *class_count*)

Create multibox head layers.

Arguments:
config {dict} – Configuration for multibox layers. class_count {int} – Number of object classes.

Returns:
tuple – Tuple of lists containing class head and location head layers.

1.6.5 models.sfdet_vgg module

class models.sfdet_vgg.**SFDetVGG**(*mode*, *base*, *extras*, *fusion_module*, *pyramid_module*, *head*, *anchors*, *class_count*)

Bases: Module

forward(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

init_weights(*model_save_path*, *base_network*)

load_weights(*base_file*)

models.sfdet_vgg.**build_SFDetVGG**(*mode*, *new_size*, *anchors*, *class_count*, *model_save_path*, *pretrained_model*, *output_txt*)

models.sfdet_vgg.**get_extras**(*in_channels*, *batch_norm=False*)

models.sfdet_vgg.**get_fusion_module**(*config*, *base*, *extras*)

models.sfdet_vgg.**get_pyramid_module**(*config*)

models.sfdet_vgg.**multibox**(*config*, *class_count*)

1.6.6 models.ssd module

class models.ssd.**SSD**(*mode*, *base*, *extras*, *head*, *anchors*, *class_count*)

Bases: Module

SSD architecture

forward(*x*)

Forward pass of the SSD model.

Arguments:
x {torch.Tensor} – Input tensor.

Returns:
torch.Tensor – Predictions.

init_weights(*model_save_path*, *basenet*)

Initialize model weights.

Arguments:
model_save_path {str} – Path to save the model. basenet {str} – Base network for the model.

load_weights(*base_file*)

Load pre-trained weights.

Arguments:
base_file {str} – Path to the pre-trained weights file.

models.ssd.**build_SSD**(*mode*, *new_size*, *anchors*, *class_count*)

Build SSD model.

Arguments:
mode {str} – Operating mode, either 'train' or 'test'. new_size {int} – Size of the input images. anchors {torch} –ensor): Anchors for bounding box predictions. class_count {int} – Number of object classes.

Returns:
SSD – SSD model.

models.ssd.**get_extras**(*config*, *in_channels*, *batch_norm=False*)

Get extra layers for SSD model.

Arguments:
config {dict} – Configuration for extra layers. in_channels {int} – Number of input channels. batch_norm {bool} – Whether to use batch normalization.

Returns:
list – List of extra layers.

models.ssd.**multibox**(*config*, *base*, *extra_layers*, *class_count*)

Create multibox head layers for SSD model.

Arguments:
config {dict} – Configuration for multibox layers. base {VGG} – Base VGG model. extra_layers {list} – List of extra layers. class_count {int} – Number of object classes.

Returns:
tuple – Tuple of lists containing class head and location head layers.

1.6.7 models.vgg module

class models.vgg.**VGG**(*config*, *in_channels*, *batch_norm=False*)

Bases: object

VGG classification architecture

get_layers()

Forms the layers of the model based on self.config

Returns:
list – contains all layers of VGG model based on self.config

1.6.8 Module contents

1.7 solver module

class solver.**Solver**(*version*, *data_loader*, *config*, *output_txt*)

Bases: object

DEFAULTS = {}

adjust_learning_rate(*optimizer*, *gamma*, *step*)

Sets the learning rate to the initial LR decayed by 10 at every
specified step

Adapted from PyTorch Imagenet example: # https://github.com/pytorch/examples/blob/master/imagenet/main.py

build_model()

Instantiate the model, loss criterion, and optimizer

eval(*dataset*, *max_per_image*, *score_threshold*)

model_step(*images*, *targets*, *count*)

A step for each iteration

print_loss_log(*start_time*, *iters_per_epoch*, *e*, *i*, *class_loss*, *loc_loss*, *loss*)

Prints the loss and elapsed time for each epoch

print_network(*model*)

Prints the structure of the network and the total number of parameters

save_model(*e*)

Saves a model per e epoch

test()

testing process

train()

training process