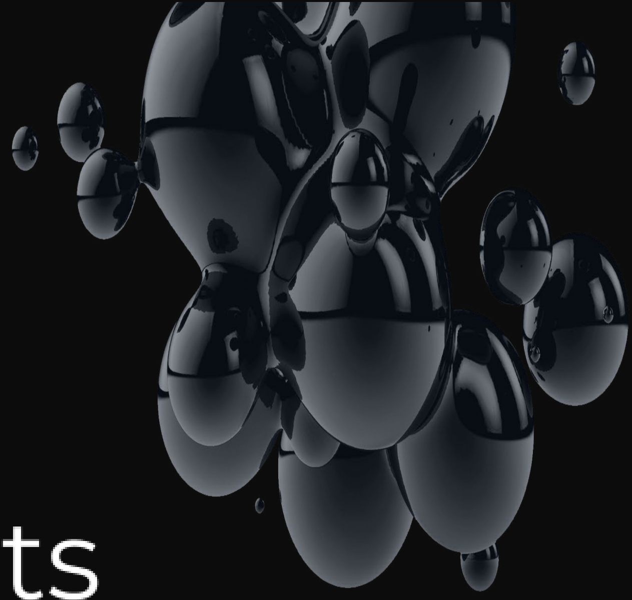


SuperGradients User Guide

Deci's Deep Learning Training Tool

Open-Source and Free

Break the AI Barrier

Version 1.0.

deci.ai

Table of Contents

Contact Information

Email – support@deci.ai

Israel
Sasson Hugi Tower, Abba Hillel Silver Rd 12,
Ramat Gan, Israel

Revision History

1.0.1	January 2021	Initial version

SuperGradients

What is SuperGradients?

The SuperGradients PyTorch-based training library provides a quick, simple and free open-source platform in which you can train your models using state of the art techniques.

Who can use SuperGradients:

- **Open Source Users –** The SuperGradients can be used to easily train your models regardless of whether you ever have or ever will use the Deci platform.
- **Deci Customers –** The SuperGradients library can reproduce the training procedure performed by Deci for their optimized models.

Introducing the SuperGradients library

The **SuperGradients** training library provides all of the scripts, example code and configurations required to demonstrate how to train your model on a dataset and to enable you to do it by yourself.

SuperGradients comes as an easily installed Python package (pip install) that you can integrate into your code base in order to train your models.

Installation

► **To install the SuperGradients library –**

1. Run the following command on your machine's terminal –

```
pip install super_gradients
```

Integrating Your Training Code - Complete Walkthrough

Whether you are a Deci customer, or an open source SuperGradients user- it is likely that you already have your own training script, model, loss function implementation etc.

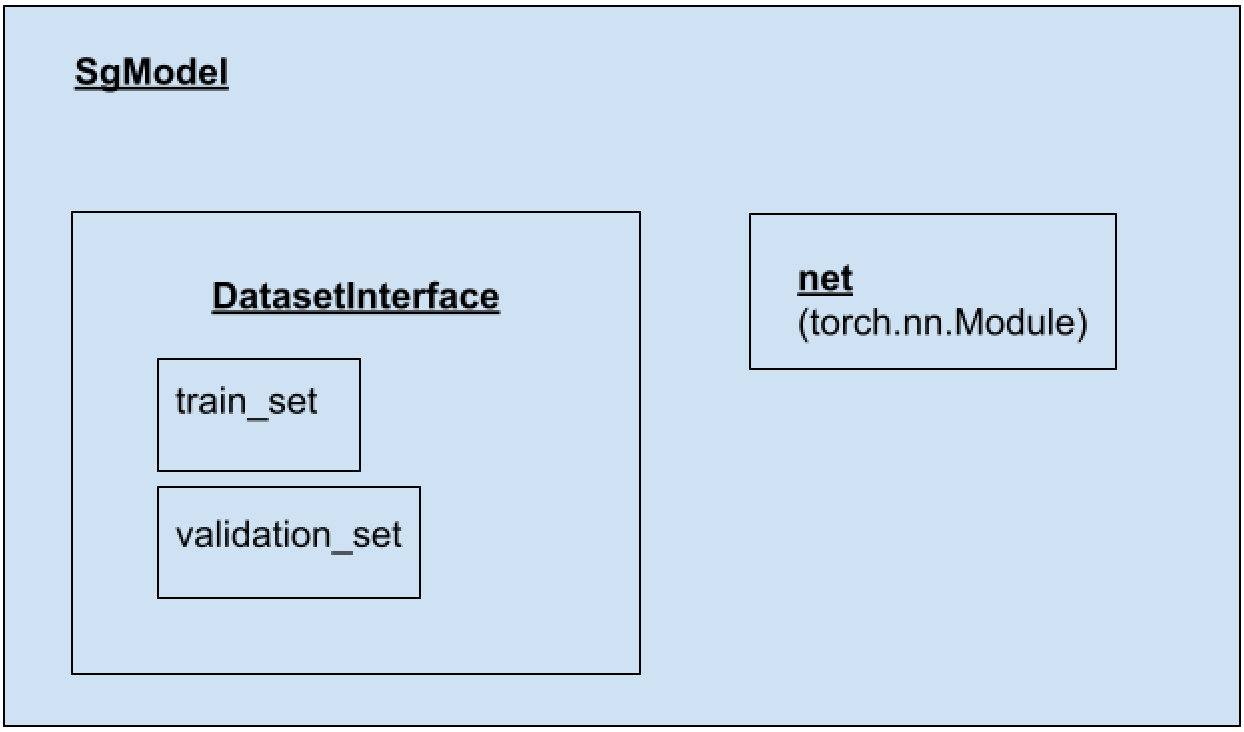
In this section we present the modifications needed in order to launch your training using SuperGradients.

Integrating Your Training Code: Main components:

SgModel - the main class in charge of training, testing, logging and basically everything that has to do with the execution of training code.

DatasetInterface - which is passed as an argument to the SgModel and wraps the training set, validation set and optionally a test set for the SgModel instance to work with accordingly.

SgModel.net -The network to be used for training/testing (of torch.nn.Module type).



Integrating Your Training Code - Complete Walkthrough: Dataset

The specified dataset interface class must inherit from **super_gradients.training.datasets.dataset_interfaces.dataset_interface**, which is where data augmentation and data loader configurations are defined.

For instance, a dataset interface for Cifar10:

```
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from super_gradients.training import utils as core_utils
from super_gradients.training.datasets.dataset_interfaces import
DatasetInterface


class UserDataset(DatasetInterface):

   def __init__(self, name="cifar10", dataset_params={}):
       super(UserDataset, self).__init__(dataset_params)
       self.dataset_name = name
       self.lib_dataset_params = {'mean': (0.4914, 0.4822, 0.4465), 'std':
(0.2023, 0.1994, 0.2010)}

       crop_size = core_utils.get_param(self.dataset_params, 'crop_size',
default_val=32)

       transform_train = transforms.Compose([
           transforms.RandomCrop(crop_size, padding=4),
           transforms.RandomHorizontalFlip(),
           transforms.ToTensor(),
           transforms.Normalize(self.lib_dataset_params['mean'],
self.lib_dataset_params['std']),
       ])

       transform_val = transforms.Compose([
           transforms.ToTensor(),
           transforms.Normalize(self.lib_dataset_params['mean'],
self.lib_dataset_params['std']),
       ])

       self.trainset = datasets.CIFAR10(root=self.dataset_params.dataset_dir,
train=True, download=True,
                                        transform=transform_train)

       self.valset = datasets.CIFAR10(root=self.dataset_params.dataset_dir,
```

```
train=False, download=True,
                                    transform=transform_val)
```

Required parameters can be passed using the `python dataset_params` argument. When implementing a dataset interface, the`trainset` and `valset` attributes are required and must be initiated with a *torch.utils.data.Dataset* type. These fields will cause the *SgModule* instance to use them accordingly, such as during training, testing, and so on.

Integrating Your Training Code - Complete Walkthrough: Model

This is rather straightforward- the only requirement is that the model must be of torch.nn.Module type. In our case, a simple Lenet implementation (taken from https://github.com/icpm/pytorch-cifar10/blob/master/models/LeNet.py).

```
import torch.nn as nn
import torch.nn.functional as func


class LeNet(nn.Module):
   def __init__(self):
       super(LeNet, self).__init__()
       self.conv1 = nn.Conv2d(3, 6, kernel_size=5)
       self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
       self.fc1 = nn.Linear(16*5*5, 120)
       self.fc2 = nn.Linear(120, 84)
       self.fc3 = nn.Linear(84, 10)

   def forward(self, x):
       x = func.relu(self.conv1(x))
       x = func.max_pool2d(x, 2)
       x = func.relu(self.conv2(x))
       x = func.max_pool2d(x, 2)
       x = x.view(x.size(0), -1)
       x = func.relu(self.fc1(x))
       x = func.relu(self.fc2(x))
       x = self.fc3(x)
       return x
```

Integrating Your Training Code - Complete Walkthrough: Loss Function

The loss function class must be of *torch.nn.module._LOSS* type. For example, our *LabelSmoothingCrossEntropyLoss* implementation.

```
import torch.nn as nn
from super_gradients.training.losses.label_smoothing_cross_entropy_loss
import cross_entropy

class LabelSmoothingCrossEntropyLoss(nn.CrossEntropyLoss):
   def __init__(self, weight=None, ignore_index=-100, reduction='mean',
smooth_eps=None, smooth_dist=None,
                from_logits=True):
       super(LabelSmoothingCrossEntropyLoss, self).__init__(weight=weight,

ignore_index=ignore_index, reduction=reduction)
       self.smooth_eps = smooth_eps
       self.smooth_dist = smooth_dist
       self.from_logits = from_logits

   def forward(self, input, target, smooth_dist=None):
       if smooth_dist is None:
           smooth_dist = self.smooth_dist
       loss = cross_entropy(input, target, weight=self.weight,
ignore_index=self.ignore_index,
                            reduction=self.reduction,
smooth_eps=self.smooth_eps,
                            smooth_dist=smooth_dist,
from_logits=self.from_logits)

       return loss
```

Important – *forward(...)* may return a (loss, loss_items) tuple instead of just a single item (i.e loss), where –
loss is the tensor used for backprop, meaning what your original loss function returns.
loss_items must be a tensor of shape (n_items) that is composed of values that are computed during the forward pass, so that it can be logged over the entire epoch.
For example, the loss itself should always be logged. Another example is a scenario where the computed loss is the sum of a few components. These entries should be logged in loss_items.

During training, set the *loss_logging_items_names* parameter in *training_params* to be a list of strings of length *n_items*, whose ith element is the name of the ith entry in loss_items. In this way, each item will be logged, rendered and monitored in TensorBoard, thus saving model checkpoints accordingly.

Because running logs save the loss_items in some internal state. It is therefore recommended that loss_items be detached from their computational graph for memory efficiency.

Integrating Your Training Code - Complete Walkthrough: Metrics

The metrics objects to be logged during training must be of torchmetrics.Metric type. For more information on how to use torchmetric.Metric objects and implement your own metrics. see https://torchmetrics.readthedocs.io/en/latest/pages/overview.html.

During training, the metric's update is called with the model's raw outputs and raw targets. Therefore, any processing of the two must be taken into account and applied in the *update*.

Training works out of the box with any of the module torchmetrics (full list in https://torchmetrics.readthedocs.io/en/latest/references/modules.html). Additional metrics implementations such as mean average precision for object detection can be found at *super_gradients.training.metrics*)

```
import torchmetrics
import torch


class Accuracy(torchmetrics.Accuracy):
   def __init__(self, dist_sync_on_step=False):
       super().__init__(dist_sync_on_step=dist_sync_on_step, top_k=1)

   def update(self, preds: torch.Tensor, target: torch.Tensor):
       super().update(preds=preds.softmax(1), target=target)


class Top5(torchmetrics.Accuracy):
   def __init__(self, dist_sync_on_step=False):
       super().__init__(dist_sync_on_step=dist_sync_on_step, top_k=5)

   def update(self, preds: torch.Tensor, target: torch.Tensor):
       super().update(preds=preds.softmax(1), target=target)
```

Integrating Your Training Code- Complete Walkthrough: Training script

We instantiate an SgModel and a UserDatasetInterface, then call connect_dataset_interface which will initialize the dataloaders and pass additional dataset parameters to the SgModel instance.

```
from super_gradients.training import SgModel

sg_model = SgModel(experiment_name='LeNet_cifar10_example')
dataset_params = {"batch_size": 256}
dataset = UserDataset(dataset_params)
sg_model.connect_dataset_interface(dataset)
```

Now, we pass a LeNet instance we defined above to the SgModel:

```
network = LeNet()
sg_model.build_model(network)
```

Next, we define metrics in order to evaluate our model.

```
 from super_gradients.training.metrics import Accuracy, Top5

train_metrics_list = [Accuracy(), Top5()]
valid_metrics_list = [Accuracy(), Top5()]
```

Initializing the loss, and specifying training parameters

```
train_params = {"max_epochs": 250,
               "lr_updates": [100, 150, 200],
               "lr_decay_factor": 0.1,
               "lr_mode": "step",
               "lr_warmup_epochs": 0,
               "initial_lr": 0.1,
```

```
                 "loss": LabelSmoothingCrossEntropyLoss(),
                 "criterion_params": {},
                 "optimizer": "SGD",
                 "optimizer_params": {"weight_decay": 1e-4, "momentum":0.9},
                 "launch_tensorboard": False,
                 "train_metrics_list": train_metrics_list,
                 "valid_metrics_list": valid_metrics_list,
                 "loss_logging_items_names": ["Loss"],
                 "metric_to_watch": "Accuracy",
                 "greater_metric_to_watch_is_better": True}

sg_model.train(train_params)
```

Training Parameter Notes:

- *loss_logging_items_names* parameter – Refers to the single item returned in *loss_items* in our loss function described above.
- *metric_to_watch* – Is the model's metric that determines the checkpoint to be saved. In our example, this parameter is set to *Accuracy*, and can be set to any of the following:
  - A metric name (str) of one of the metric objects from the *valid_metrics_list*.
  - A *metric_name* that represents a metric that appears in *valid_metrics_list* and has an attribute *component_names*. *component_names* is a list that refers to the names of each entry in the output metric (torch tensor of size n).
  - One of the *loss_logging_items_names*, such as one that corresponds to an item returned during the loss function's forward pass as discussed earlier.
- *greater_metric_to_watch_is_better* flag – Determines when to save a model's checkpoint according to the value of the `metric_to_watch`.

Training Parameters

The following is a description of all the parameters passed in *training_params* when *train()* is called.

`max_epochs`: int

Number of epochs to run during training.

`lr_updates`: list(int)

List of fixed epoch numbers to perform learning rate updates when `lr_mode='step'`.

`lr_decay_factor`: float

Decay factor to apply to the learning rate at each update when *lr_mode='step'*.

`lr_mode`: str

Learning rate scheduling policy, one of ['step','poly','cosine','function'].

- 'step' refers to constant updates of epoch numbers passed through `lr_updates`.
- 'cosine' refers to Cosine Annealing policy as described in https://arxiv.org/abs/1608.03983.
- 'poly' refers to polynomial decrease, such as in each epoch iteration `self.lr = self.initial_lr * pow((1.0 - (current_iter / max_iter)), 0.9)`
- 'function' refers to a user defined learning rate scheduling function, that is passed through `lr_schedule_function`.

`lr_schedule_function`: Union[callable,None]

Learning rate scheduling function to be used when `lr_mode` is 'function'.

`lr_warmup_epochs`: int (default=0)

Number of epochs for learning rate warm up. For more information, you may refer to https://arxiv.org/pdf/1706.02677.pdf (Section 2.2).

`cosine_final_lr_ratio`: float (default=0.01)

Final learning rate ratio (only relevant when `lr_mode`='cosine'). The cosine starts from initial_lr and reaches initial_lr * cosine_final_lr_ratio in the last epoch.

`inital_lr`: float

Initial learning rate.

`loss`: Union[nn.module, str]

Loss function to be used for training.

One of super_gradients's built in options:

"cross_entropy": LabelSmoothingCrossEntropyLoss,

"mse": MSELoss,

"r_squared_loss": RSquaredLoss,

"detection_loss": YoLoV3DetectionLoss,

"shelfnet_ohem_loss": ShelfNetOHEMLoss,

"shelfnet_se_loss": ShelfNetSemanticEncodingLoss,

"yolo_v5_loss": YoLoV5DetectionLoss,

"ssd_loss": SSDLoss,

or user defined nn.module loss function.

Important – *forward(…)* should return a (loss, loss_items) tuple, where –

- *loss* is the tensor used for backprop, meaning what your original loss function returns
- *loss_items* must be a tensor of shape (n_items) of values computed during the forward pass, so that they can be logged over the entire epoch.

For example, the loss itself should always be logged. Another example is a scenario where the computed loss is the sum of a few components. These entries should be returned in loss_items.

During training, set the *loss_logging_items_names* parameter in *training_params* to be a list of strings of length *n_items*, whose ith element is the name of the ith entry in loss_items. In this way, each item will be logged, rendered on TensorBoard and monitored, thus saving model checkpoints accordingly.

Running logs saves the loss_items in some internal state. It is therefore recommended that loss_items be detached from their computational graph for memory efficiency.

`optimizer`: str

Optimization algorithm. One of ['Adam','SGD','RMSProp'] corresponding to the torch.optim optimzer implementations.

`criterion_params`: dict

Loss function parameters.

`optimizer_params`: dict

Optimizer parameters. You may refer to https://pytorch.org/docs/stable/optim.html for the full list of the parameters for each optimizer.

`train_metrics_list`: list(torchmetrics.Metric)

Metrics to log during training. You may refer to https://torchmetrics.rtfd.io/en/latest/, for more information about TorchMetrics.

`valid_metrics_list`: list(torchmetrics.Metric)

Metrics to log during validation/testing. You may refer to https://torchmetrics.rtfd.io/en/latest/, for more information about TorchMetrics.

`loss_logging_items_names`: list(str)

The list of names/titles for the outputs returned from the loss function's forward pass. These names are used to log their values.
Note – The loss function should return the tuple (loss, loss_items).

`metric_to_watch`: str (default="Accuracy")

Specifies the metric according to which the model checkpoint is saved. It can be set to any of the following:

- A metric name (str) of one of the metric objects from the valid_metrics_list

- A "metric_name" to be used if any metric in the valid_metrics_list has an attribute component_names, which is a list referring to the names of each entry in the output metric (torch tensor of size n).
- One of the "loss_logging_items_names" `that` corresponds to an item to be returned during the loss function's forward pass.

At the end of each epoch, if a new best *metric_to_watch* value is achieved, the model's checkpoint is saved in YOUR_PYTHON_PATH/checkpoints/ckpt_best.pth.

`greater_metric_to_watch_is_better`: bool

Determines when to save a model's checkpoint according to the value of the `metric_to_watch`:

- *True:* A model's checkpoint is saved when the model achieves the highest metric_to_watch.
- *False:* A model's checkpoint is saved when the model achieves the lowest metric_to_watch.

`ema`: bool (default=False)

Specifies whether to use Model Exponential Moving Average. You may refer to https://github.com/rwightman/pytorch-image-models ema implementation), for more information.

`batch_accumulate`: int (default=1)

Number of batches to accumulate before every backward pass.

`ema_params`: dict

Parameters for the ema model.

`zero_weight_decay_on_bias_and_bn`: bool (default=False)

Specifies whether to apply weight decay on batch normalization parameters or not.

`load_opt_params`: bool (default=True)

Specifies whether to load the optimizers parameters (as well) when loading a model's checkpoint.

`run_validation_freq`: int (default=1)

The frequency at which validation is performed during training. This means that the validation is run every `run_validation_freq` epochs.

`save_model`: bool (default=True)

Specifies whether to save the model's checkpoints.

`launch_tensorboard`: bool (default=False)

Specifies whether to launch a TensorBoard process.

`tb_files_user_prompt`: bool

Displays the TensorBoard deletion user prompt.

`silent_mode`: bool

Deactivates the printouts.

`mixed_precision`: bool

Specifies whether to use mixed precision or not.

`tensorboard_port`: int, None (default=None)

Specific port number for the TensorBoard to use when launched (when set to None, some free port number will be used).

`save_ckpt_epoch_list`: list(int) (default=[])

Specifies the list of fixed epoch indices in which to save checkpoints.

`average_best_models`: bool (default=False)

If True, a snapshot dictionary file and the average model will be saved / updated at every epoch and only evaluated after the training has completed. The snapshot file will only be deleted upon completing the training. The snapshot dict will be managed on the CPU.

`save_tensorboard_to_s3`: bool (default=False)

If True, saves the TensorBoard in S3.

`precise_bn`: bool (default=False)

Whether to use precise_bn calculation during the training.

`precise_bn_batch_size`: int (default=None)

The effective batch size we want to calculate the batchnorm on. For example, if we are training a model on 8 gpus, with a batch of 128 on each gpu, a good rule of thumb would be to give it 8192 (ie: effective_batch_size * num_gpus = batch_per_gpu * m_gpus * num_gpus). If precise_bn_batch_size is not provided in the training_params, the latter heuristic will be taken.

`seed` : int (default=42)

Random seed to be set for torch, numpy, and random. When using DDP each process will have it's seed set to seed + rank.

`log_installed_packages`: bool (default=False)

When set, the list of all installed packages (and their versions) will be written to the tensorboard and logfile (useful when trying to reproduce results).

`dataset_statistics`:: bool (default=False)

Enable a statistic analysis of the dataset. If set to True the dataset will be analyzed and a report will be added to the tensorboard along with some sample images from the dataset. Currently only detection datasets are supported for analysis.

`save_full_train_log` : bool (default=False)

When set, a full log (of all super_gradients modules, including uncaught exceptions from any other module) of the training will be saved in the checkpoint directory under full_train_log.log

Logs and Checkpoints

The model's weights, logs and tensorboards are saved in *"YOUR_PYTHONPATH"/ checkpoints/"YOUR_EXPERIMENT_NAME"* . (In our walkthrough example, *"YOUR_EXPERIMENT_NAME"* is *user_model_training).*

► To watch training progress –

1st option:

1. Open a terminal.
2. Navigate to *"YOUR_LOCAL_PATH_TO_super_gradients_PACKAGE"/* and run `` `tensorboard --logdir checkpoints --bind_all``.

   The message `TensorBoard 2.4.1 at http://localhost:XXXX/` appears.
3. Follow the link in this message to see the progress of the training.

2nd option:

Set the "launch_tensorboard_process" flag in your training_params passed to SgModel.train(...), and follow instructions displayed in the shell.

► To resume training –

When building the network- call SgModel.build_model(...load_checkpoint=True). Doing so, will load the network's weights, as well as any relevant information for resuming training (monitored metric values, optimizer states, etc) with the latest checkpoint. For more advanced usage see SgModel.build_model docs in code.

- **Checkpoint structure – state_dict (see https://pytorch.org/tutorials/beginner/saving_loading_models.html for more information regarding state_dicts) with the following keys:**

  -"net"- The network's state_dict.

  -"acc"- The value of `metric_to_watch` from training.

  -"epoch"- Last epoch performed before saving this checkpoint.

  -"ema_net" [Optionall, exists if training was performed with EMA] -

  The state dict of the EMA net.

  -"optimizer_state_dict"- Optimizer's state dict from training.

  -"scaler_state_dict"- Gradient scalar state_dict from training.

Dataset Parameters

dataset_params argument passed to SgModel.build_model().

`batch_size`: int (default=64)

Number of examples per batch for training. Large batch sizes are recommended.

`test_batch_size`: int (default=200)

Number of examples per batch for test/validation. Large batch sizes are recommended.

`dataset_dir`: str (default="./data/")

Directory location for the data. Data will be downloaded to this directory when received from a remote URL.

`s3_link`: str (default=None)

The remote s3 link from which to download the data (optional).

Network Architectures

The following architectures are implemented in SuperGradients' code, and can be initialized by passing their name (i.e string) to SgModel.build_model easily.

For example:

```
sg_model = SgModel("resnet50_experiment")
sg_model.build_model(architecture="resnet50")
```

Will initialize a resnet50 and set it to be sg_model's network attribute, which will be used for training.

'resnet18',

'resnet34',

'resnet50_3343',

'resnet50',

'resnet101',

'resnet152',

'resnet18_cifar',

'custom_resnet',

'custom_resnet50',

'custom_resnet_cifar',

'custom_resnet50_cifar',

'mobilenet_v2',

'mobile_net_v2_135',

'custom_mobilenet_v2',

'mobilenet_v3_large',

'mobilenet_v3_small',

'mobilenet_v3_custom',

'yolo_v3',

'tiny_yolo_v3',

'custom_densenet',

'densenet121',

'densenet161',

'densenet169',

'densenet201',

'shelfnet18',

'shelfnet34',

'shelfnet50_3343',

'shelfnet50',

'shelfnet101',

'shufflenet_v2_x0_5',

'shufflenet_v2_x1_0',

'shufflenet_v2_x1_5',

'shufflenet_v2_x2_0',

'shufflenet_v2_custom5',

'darknet53',

'csp_darknet53',

'resnext50',

'resnext101',

'googlenet_v1',

'efficientnet_b0',

'efficientnet_b1',

'efficientnet_b2',

'efficientnet_b3',

'efficientnet_b4',

'efficientnet_b5',

'efficientnet_b6',

'efficientnet_b7',

'efficientnet_b8',

'efficientnet_l2',

'CustomizedEfficientnet',

'regnetY200',

'regnetY400',

'regnetY600',

'regnetY800',

'custom_regnet',

'nas_regnet',

'yolo_v5s',

'yolo_v5m',

'yolo_v5l',

'yolo_v5x',

'custom_yolov5',

'ssd_mobilenet_v1',

'ssd_lite_mobilenet_v2',

'repvgg_a0',

'repvgg_a1',

'repvgg_a2',

'repvgg_b0',

'repvgg_b1',

'repvgg_b2',

'repvgg_b3',

'repvgg_d2se',

'repvgg_custom'

Pretrained Models

Classification models

Model	Dataset	arch_params	Top-1	Latency b1 T4
EfficientNet B0	ImageNet		77.62	1.16ms
RegNetY200	ImageNet		70.88	-
RegNetY400	ImageNet		74.74	-
RegNetY600	ImageNet		76.18	-
RegNetY800	ImageNet		77.07	-

ResNet18	ImageNet		70.6	0.599ms
ResNet34	ImageNet		74.13	0.89ms
ResNet50	ImageNet	{"pretrained_weights": "imagenet", “num_classes”:1000}	76.3	0.94ms
MobileNetV3_large-150 epochs	ImageNet		73.79	0.87ms
MobileNetV3_large-300 epochs	ImageNet		74.52	0.87ms
MobileNetV3_small	ImageNet		67.45	0.75ms
MobileNetV2_w1	ImageNet		73.08	0.58ms

Object Detection models

Model	Dataset	arch_params	mAPval 0.5:0.95	Latency b1T4	Throughout b64T4
YOLOv5 small	CoCo	640x640	37.3	10.09ms	101.85fps
YOLOv5 medium	CoCo	640x640	45.2	17.55ms	57.66fps

Semantic Segmentation models

Model	Dataset	arch_params	mIoU	Latency b1T4	Throughout b64T4
DDRNet23	Cityscapes		78.65	-	-
DDRNet23	Cityscapes		76.6		

slim					

Example- how to load a pretrained model:

```
sg_model = SgModel("resnet50_experiment")

sg_model.build_model(architecture="resnet50",
                      arch_params={"pretrained_weights": "imagenet",
"num_classes": 1000}
                      )
```

How to reproduce our training recipes:

The training recipes for the pretrained models are completely visible for the SuperGradients' users and can be found under *"YOUR_LOCAL_PATH_TO_SUPER_GRADIENTS_PACKAGE"/ examples/{DATASET_NAME}_{ARCHITECTURE_NAME}_example.*

The corresponding YAML configuration files can be found under "YOUR_LOCAL_PATH_TO_SUPER_GRADIENTS_PACKAGE"/conf/{DATASET_NAME}_{ARCHITECTURE_NAME}_conf

The configuration files include the specific instructions on how to run the training recipes for reproducibility, as well as links to our tensorboards and logs from their training. Additional information regarding training time, metric scores on different configurations can be found in the configuration files as comments as well.

SuperGradients FAQ

What Type of Tasks Does the SuperGradients Support?

- Classification
- Object Detection
- Segmentation

About Deci

Artificial intelligence lies at the core of the fourth Industrial Revolution. Advanced technologies such as AI are impacting humankind more than ever before. Our mission is to enable AI developers and engineers to focus on what they do best – solving our world's most complex problems. At the same time, we at Deci challenge ourselves with how to enable more and more of these machine learning and deep learning models to fully perform in production and fulfill their true potential.

At Deci, we took an innovative approach to this challenge, using AI itself to craft the next generation of deep learning. We developed an algorithmic-first approach, focused on improving the efficacy of AI algorithms, delivering to our customers models that outperform the advantages of any other hardware or software optimization technologies.