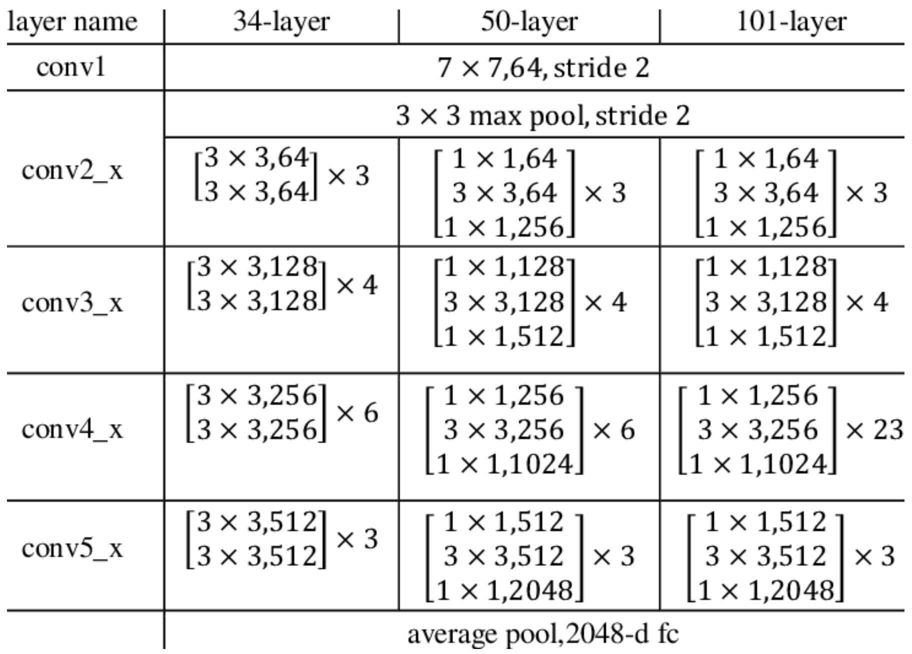
Machine Learning Presets

Resnet

- The architecture consists of “super-layers” with sets of blocks which may be regular or bottleneck.
- Nets with less than 50 layers use basic blocks which just do 3x3 filter convolutions followed by batch normalization.
- For resnet50 and above the bottleneck architecture is used which does 1x1 convolutions to transform the image filtering dimensions, first its reduced and after the main convolution layer (3x3 filter) its increased – The idea is to have only relevant (Strong) features prevail as less information is kept so noise is reduced.



layer name	34-layer	50-layer	101-layer
conv1	$7\times7, 64$, stride 2		
	3×3 max pool, stride 2		
conv2_x	$\begin{bmatrix}3\times3,64\\3\times3,64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1,64\\3\times3,64\\1\times1,256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1,64\\3\times3,64\\1\times1,256\end{bmatrix}\times3$
conv3_x	$\begin{bmatrix}3\times3,128\\3\times3,128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1,128\\3\times3,128\\1\times1,512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1,128\\3\times3,128\\1\times1,512\end{bmatrix}\times4$
conv4_x	$\begin{bmatrix}3\times3,256\\3\times3,256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1,256\\3\times3,256\\1\times1,1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1,256\\3\times3,256\\1\times1,1024\end{bmatrix}\times23$
conv5_x	$\begin{bmatrix}3\times3,512\\3\times3,512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1,512\\3\times3,512\\1\times1,2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1,512\\3\times3,512\\1\times1,2048\end{bmatrix}\times3$
	average pool, 2048-d fc		

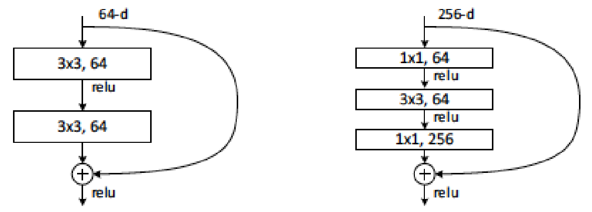


Figure 5. A deeper residual function $\mathcal{F}$ for ImageNet. Left: a building block (on 56×56 feature maps) as in Fig. 3 for ResNet-34. Right: a “bottleneck” building block for ResNet-50/101/152.

Resnet Implementation

- The official pytorch resnet implementation is used.
- The weights are pre-downloaded as a .pth and fed into the model with torch.load() and load_state_dict().
  - Init: Desired block type and architecture.
- Defaults to batch norm, there is an option to replace stride with dilation.
- Gaussian distribution weight initializations.
- Init: Weights=1 and Bias=0 for batch & group norm.
- If zero-init resolved is set to true the last batch norm weights are initialized to zero.
  - Creates layers with _make_layer.
  - FWD: Runs RN architecture, flattens tensor into vector, runs through FCL and returns vector

Functions

- Make Layer

  - Returns nn.Sequential of:

  1. Inputted blocktype “blocks” times with “planes” # of filters.

  2. Uses specified normalization.

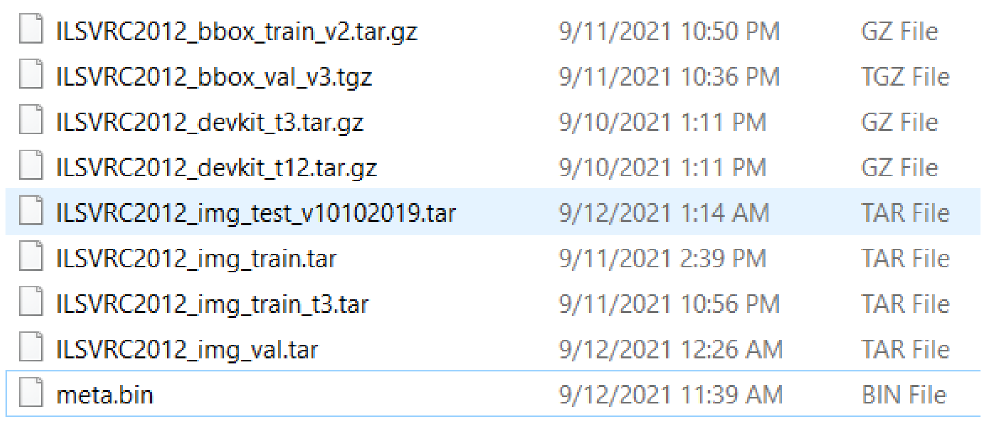
  3. Does downsampling when necessary.

  - Uses BasicBlock & Bottleneck classes for each block of layers.

- The Bottleneck and BasicBlock classes are torch.nn modules that have a forward definition with the specified layers and tensor dimensions.
- Helper functions are implemented for 1x1 and 3x3 convolutions that just return a torch.nn convolutional layer with specified parameters.

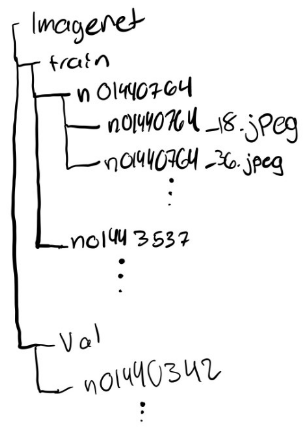
Imagenet Dataset

- Official download contains .tar and .tar.gz files.
- Torchvision has a datasets class with for Imagenet which both preprocesses the dataset and works as a dataset class to pass into the dataloader.
- The first time running the Imagenet class it will process the download files into the desired folder structure, the process takes a few hours.

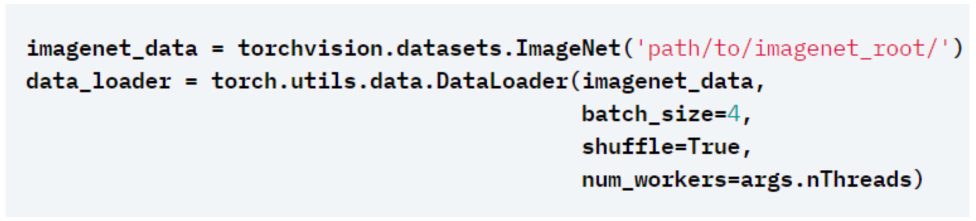


ILSVRC2012_bbox_train_v2.tar.gz	9/11/2021 10:50 PM	GZ File
ILSVRC2012_bbox_val_v3.tgz	9/11/2021 10:36 PM	TGZ File
ILSVRC2012_devkit_t3.tar.gz	9/10/2021 1:11 PM	GZ File
ILSVRC2012_devkit_t12.tar.gz	9/10/2021 1:11 PM	GZ File
ILSVRC2012_img_test_v10102019.tar	9/12/2021 1:14 AM	TAR File
ILSVRC2012_img_train.tar	9/11/2021 2:39 PM	TAR File
ILSVRC2012_img_train_t3.tar	9/11/2021 10:56 PM	TAR File
ILSVRC2012_img_val.tar	9/12/2021 12:26 AM	TAR File
meta.bin	9/12/2021 11:39 AM	BIN File

Download Files



Processed File Structure



```
imagenet_data = torchvision.datasets.ImageNet('path/to/imagenet_root/')
data_loader = torch.utils.data.DataLoader(imagenet_data,
                                          batch_size=4,
                                          shuffle=True,
                                          num_workers=args.nThreads)
```

Torchvision Imagenet Dataset Class Call

Dataloader

- Takes in a dictionary of arguments that are passed into each dataloader.
  - patch_size: model input image dimensions,
  - batch_size: images trained per batch,
  - workers: number of threads
- Creates the dataset for training and validation with the torchvision.datasets.Imagenet() prebuilt class, and passes them their respective dataloaders.

Some references

- Library used for interfacing with program

https://docs.python.org/3/library/argparse.html

- Github with a lot of examples

https://github.com/MrtnMndt/Deep_Openset_Recognition_through_Uncertainty/

- Imagenet Setup for virtual machines

https://cloud.google.com/tpu/docs/imagenet-setup

- Imagenet download script with personalizable configurations!

https://towardsdatascience.com/how-to-scrape-the-imagenet-f309e02de1f4

- Basic Imagenet use Tutorial

https://towardsdatascience.com/how-to-train-cnns-on-imagenet-ab8dd48202a9

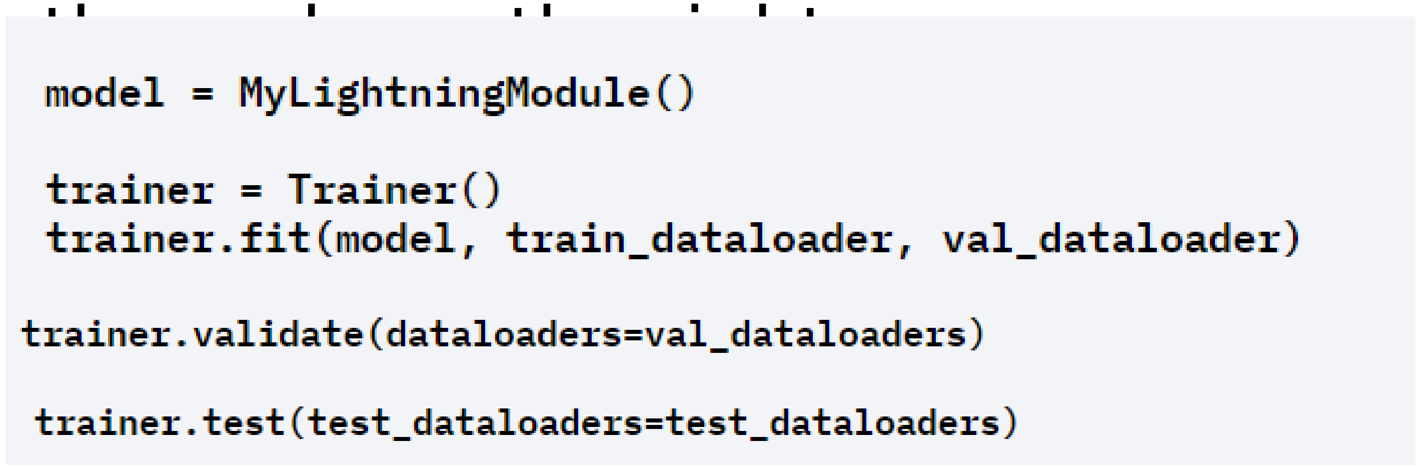
Training

Dynamic LightningModule Implementation

- https://github.com/PyTorchLightning/pytorch-lightning/blob/master/pl_examples/domain_templates/computer_vision_fine_tuning.py

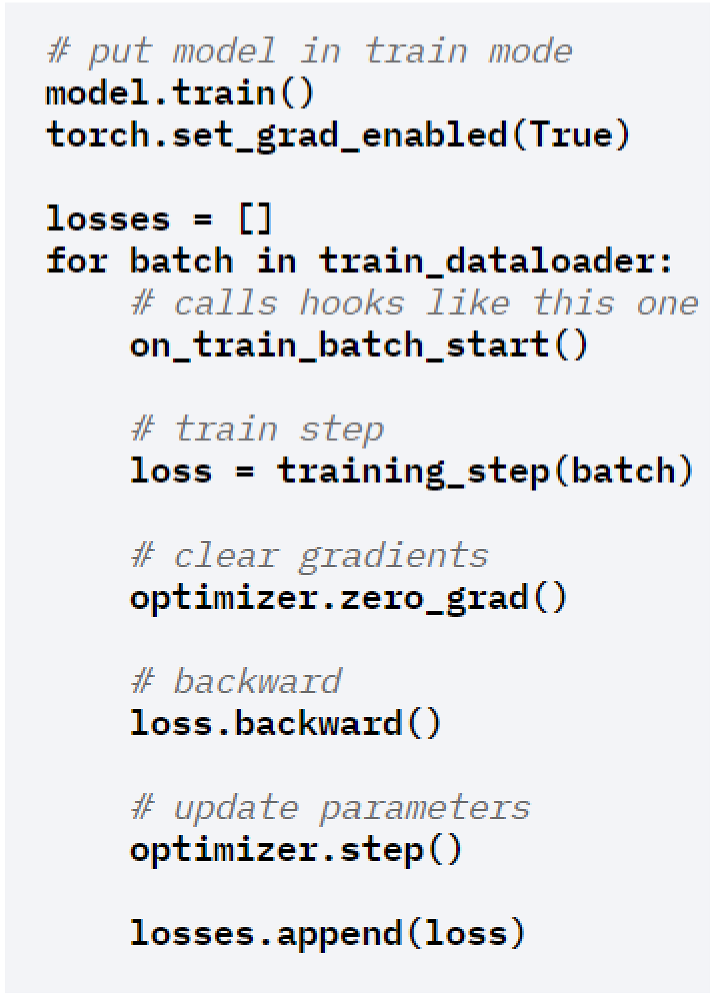
Trainer

- Once the model is organized into a lightning module, it can be trained with a trainer by calling it as below.
- The trainer class handles the training loop and details, it works as



```python
model = MyLightningModule()

trainer = Trainer()
trainer.fit(model, train_dataloader, val_dataloader)

trainer.validate(dataloaders=val_dataloaders)

trainer.test(test_dataloaders=test_dataloaders)
```

Using the trainer



```python
# put model in train mode
model.train()
torch.set_grad_enabled(True)

losses = []
for batch in train_dataloader:
    # calls hooks like this one
    on_train_batch_start()

    # train step
    loss = training_step(batch)

    # clear gradients
    optimizer.zero_grad()

    # backward
    loss.backward()

    # update parameters
    optimizer.step()

    losses.append(loss)
```

Trainer code under the hood

Train vs Validation vs Test datasets

- The **Training** set is used to fit the model.
- The **Validation** set is used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters (architecture, functions, etc.). The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration.
- The **Testing** set is used to provide an unbiased evaluation of a final model fit on the training dataset.

The validation set is like a pre-testing set, because we don’t want the model to be biased on the test set as that is the final evaluation.

Kaiming He Initialization

- Uniform

```
torch.nn.init.kaiming_uniform_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu')
```

$$\text{bound} = \text{gain} \times \sqrt{\frac{3}{\text{fan\_mode}}}$$

fan-in is the number of inputs to a hidden unit (*in this simplified diagram*, 4)

fan-out is the number of outputs to a hidden unit (*in this simplified diagram*, 4)

- Normal

```
torch.nn.init.kaiming_normal_(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu')
```

$$\text{std} = \frac{\text{gain}}{\sqrt{\text{fan\_mode}}}$$

https://medium.com/@shoray.goel/kaiming-he-initialization-a8d9ed0b5899

Glorot/Xavier Initialization

- Uniform

```
torch.nn.init.xavier_uniform_(tensor, gain=1.0)
```

$$a = \text{gain} \times \sqrt{\frac{6}{\text{fan\_in} + \text{fan\_out}}}$$

fan-in is the number of inputs to a hidden unit (*in this simplified diagram*, 4)

fan-out is the number of outputs to a hidden unit (*in this simplified diagram*, 4)

https://stackoverflow.com/questions/42670274/how-to-calculate-fan-in-and-fan-out-in-xavier-initialization-for-neural-networks

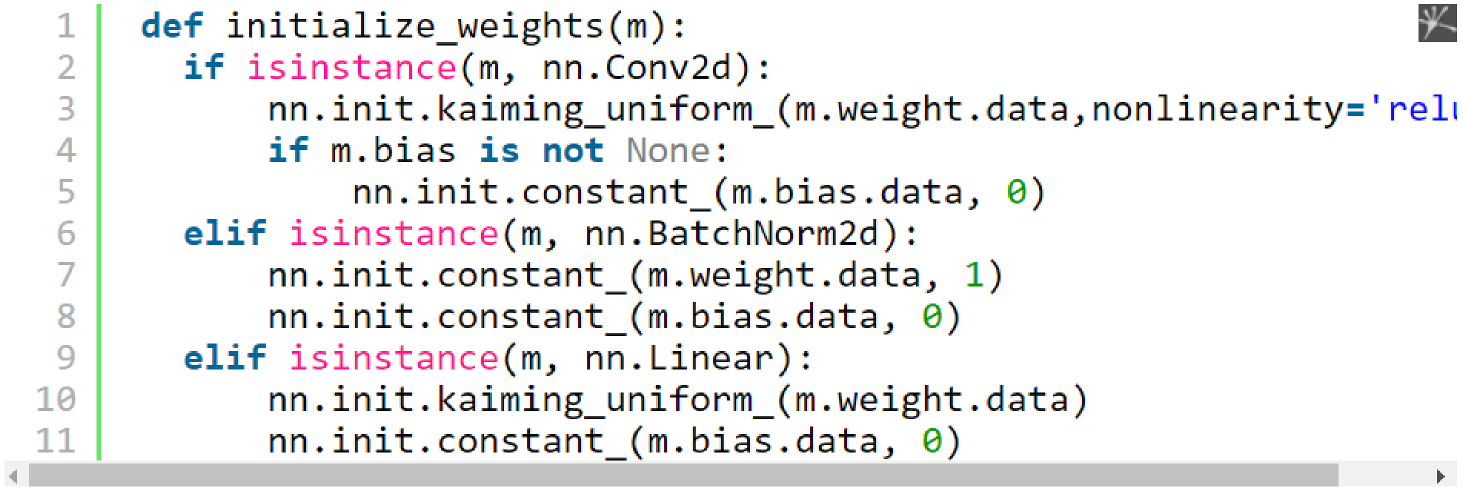
- Normal

```
torch.nn.init.xavier_normal_(tensor, gain=1.0)
```

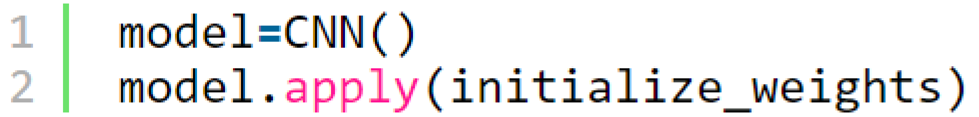
$$\text{std} = \text{gain} \times \sqrt{\frac{2}{\text{fan\_in} + \text{fan\_out}}}$$

Weight initialization implementation

- https://androidkt.com/initialize-weight-bias-pytorch/



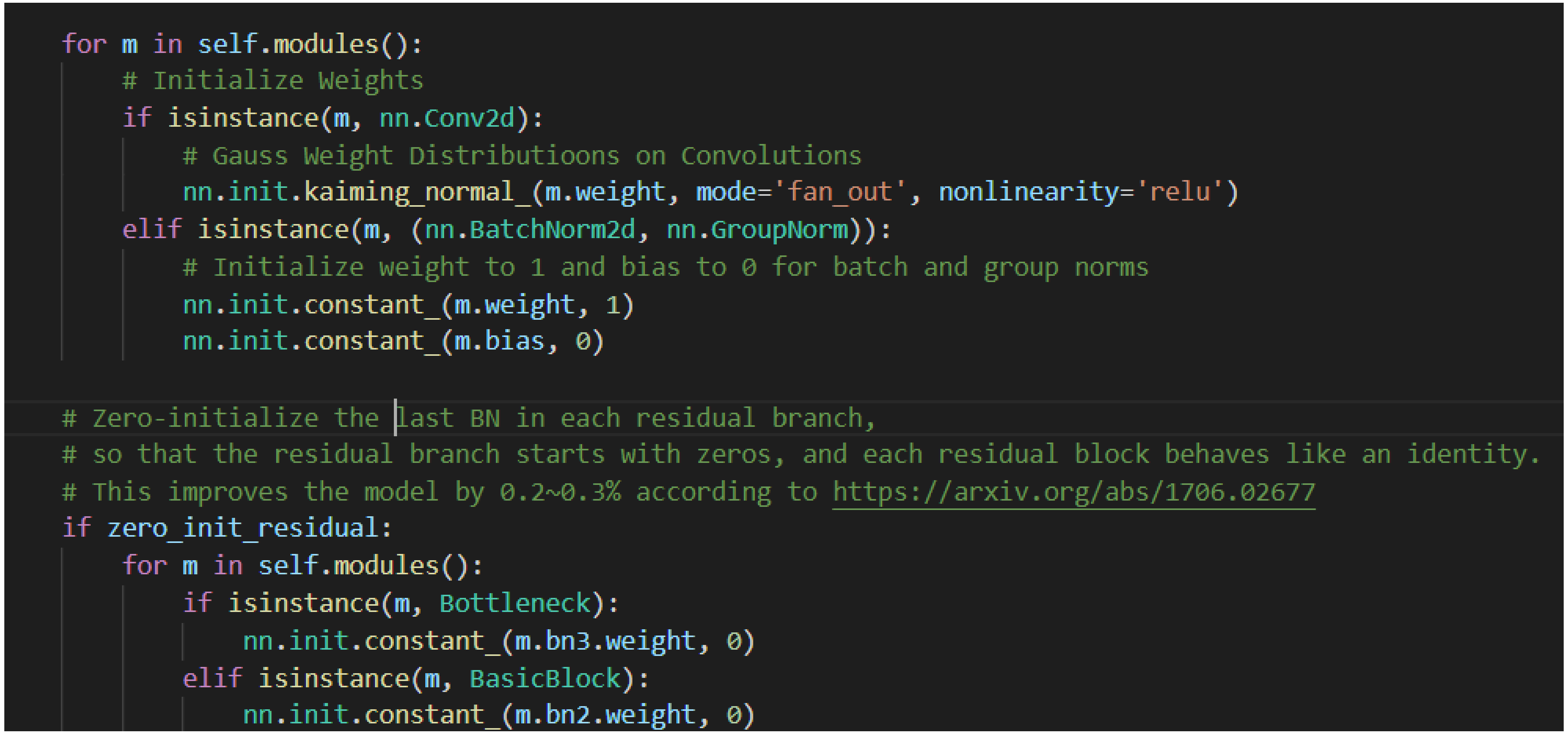
```
def initialize_weights(m):
  if isinstance(m, nn.Conv2d):
      nn.init.kaiming_uniform_(m.weight.data,nonlinearity='rel
      if m.bias is not None:
          nn.init.constant_(m.bias.data, 0)
  elif isinstance(m, nn.BatchNorm2d):
      nn.init.constant_(m.weight.data, 1)
      nn.init.constant_(m.bias.data, 0)
  elif isinstance(m, nn.Linear):
      nn.init.kaiming_uniform_(m.weight.data)
      nn.init.constant_(m.bias.data, 0)
```



```
model=CNN()
model.apply(initialize_weights)
```

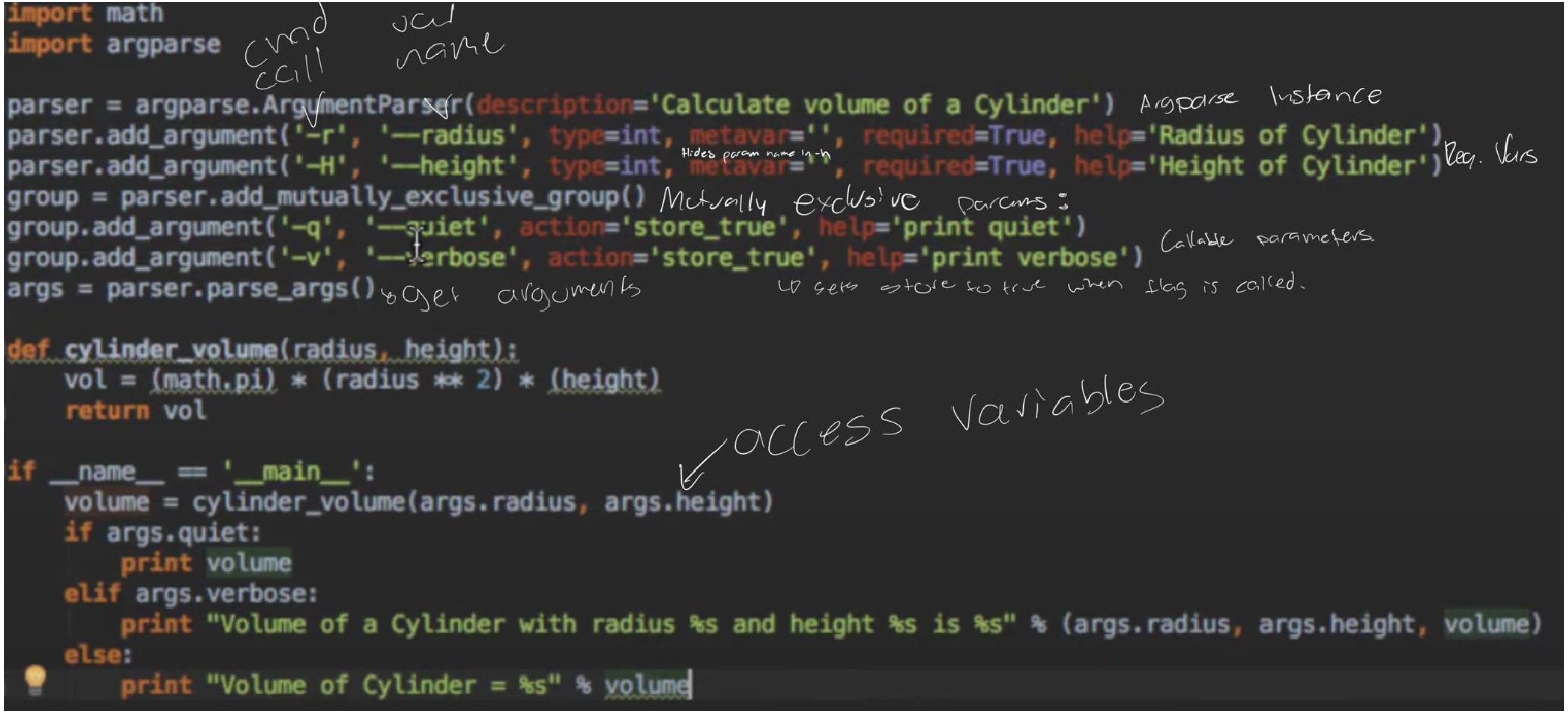
Apply works recursively onto every operation

Resnet Weight Initialization



```
for m in self.modules():
    # Initialize Weights
    if isinstance(m, nn.Conv2d):
        # Gauss Weight Distributioons on Convolutions
        nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
    elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        # Initialize weight to 1 and bias to 0 for batch and group norms
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)

# Zero-initialize the last BN in each residual branch,
# so that the residual branch starts with zeros, and each residual block behaves like an identity.
# This improves the model by 0.2~0.3% according to https://arxiv.org/abs/1706.02677
if zero_init_residual:
    for m in self.modules():
        if isinstance(m, Bottleneck):
            nn.init.constant_(m.bn3.weight, 0)
        elif isinstance(m, BasicBlock):
            nn.init.constant_(m.bn2.weight, 0)
```

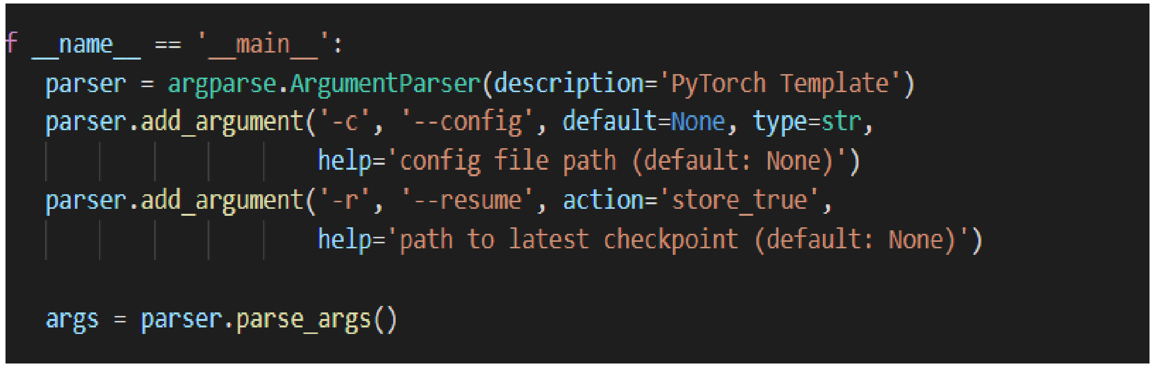
Argparse



```python
import math
import argparse

parser = argparse.ArgumentParser(description='Calculate volume of a Cylinder')
parser.add_argument('-r', '--radius', type=int, metavar='', required=True, help='Radius of Cylinder')
parser.add_argument('-H', '--height', type=int, metavar='', required=True, help='Height of Cylinder')
group = parser.add_mutually_exclusive_group()
group.add_argument('-q', '--quiet', action='store_true', help='print quiet')
group.add_argument('-v', '--verbose', action='store_true', help='print verbose')
args = parser.parse_args()

def cylinder_volume(radius, height):
    vol = (math.pi) * (radius ** 2) * (height)
    return vol

if __name__ == '__main__':
    volume = cylinder_volume(args.radius, args.height)
    if args.quiet:
        print volume
    elif args.verbose:
        print "Volume of a Cylinder with radius %s and height %s is %s" % (args.radius, args.height, volume)
    else:
        print "Volume of Cylinder = %s" % volume
```

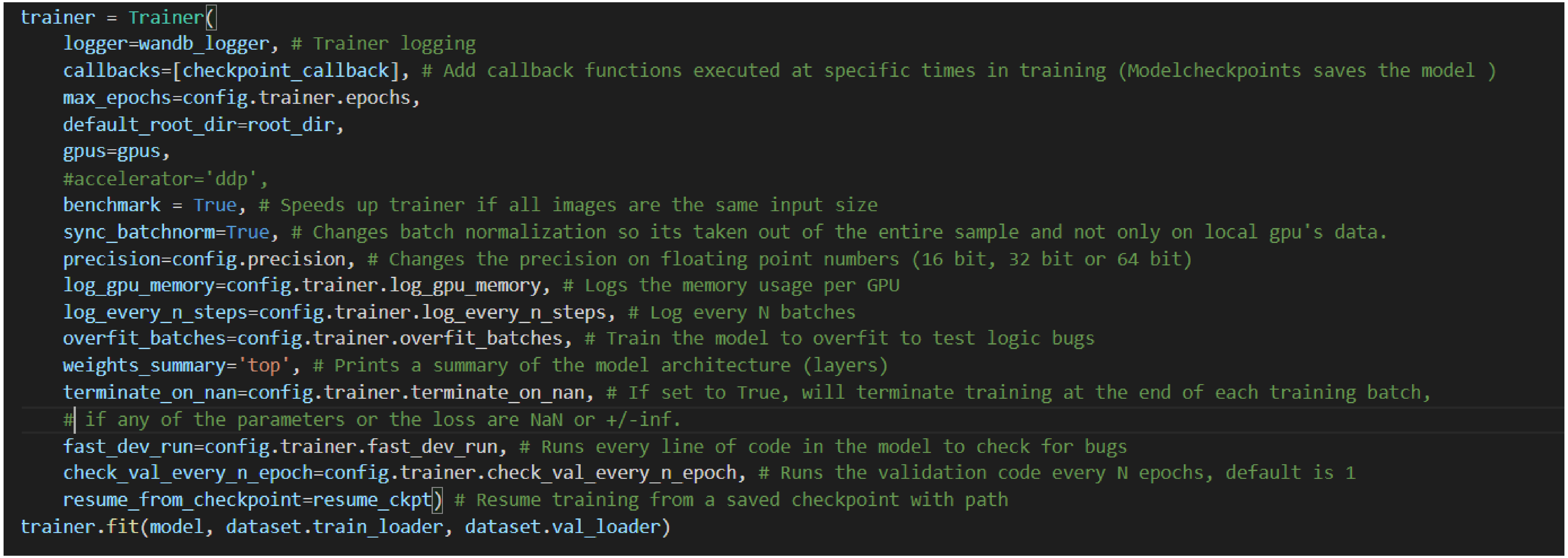
Argparse Parameters

- Only the config file path and the latest checkpoint path are handled through argparse, every other modifiable parameter is passed in through the config file.
- The checkpoint on resume can include a config file.



```
f __name__ == '__main__':
    parser = argparse.ArgumentParser(description='PyTorch Template')
    parser.add_argument('-c', '--config', default=None, type=str,
                        help='config file path (default: None)')
    parser.add_argument('-r', '--resume', action='store_true',
                        help='path to latest checkpoint (default: None)')

    args = parser.parse_args()
```

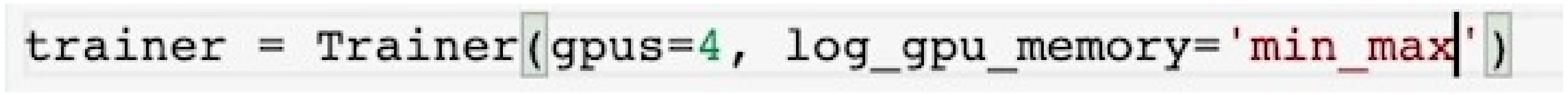
Trainer Parameters



```python
trainer = Trainer(
    logger=wandb_logger, # Trainer logging
    callbacks=[checkpoint_callback], # Add callback functions executed at specific times in training (Modelcheckpoints saves the model )
    max_epochs=config.trainer.epochs,
    default_root_dir=root_dir,
    gpus=gpus,
    #accelerator='ddp',
    benchmark = True, # Speeds up trainer if all images are the same input size
    sync_batchnorm=True, # Changes batch normalization so its taken out of the entire sample and not only on local gpu's data.
    precision=config.precision, # Changes the precision on floating point numbers (16 bit, 32 bit or 64 bit)
    log_gpu_memory=config.trainer.log_gpu_memory, # Logs the memory usage per GPU
    log_every_n_steps=config.trainer.log_every_n_steps, # Log every N batches
    overfit_batches=config.trainer.overfit_batches, # Train the model to overfit to test logic bugs
    weights_summary='top', # Prints a summary of the model architecture (layers)
    terminate_on_nan=config.trainer.terminate_on_nan, # If set to True, will terminate training at the end of each training batch,
    # if any of the parameters or the loss are NaN or +/-inf.
    fast_dev_run=config.trainer.fast_dev_run, # Runs every line of code in the model to check for bugs
    check_val_every_n_epoch=config.trainer.check_val_every_n_epoch, # Runs the validation code every N epochs, default is 1
    resume_from_checkpoint=resume_ckpt) # Resume training from a saved checkpoint with path
trainer.fit(model, dataset.train_loader, dataset.val_loader)
```

Log GPU Memory

- Logs the memory usage per GPU
- May slow training down
- Recommendation is to set flag to 1, train for a small time, tune optimization parameters and then run the full training.
- Lightning doesn’t log the memory across all nodes for performance reasons.

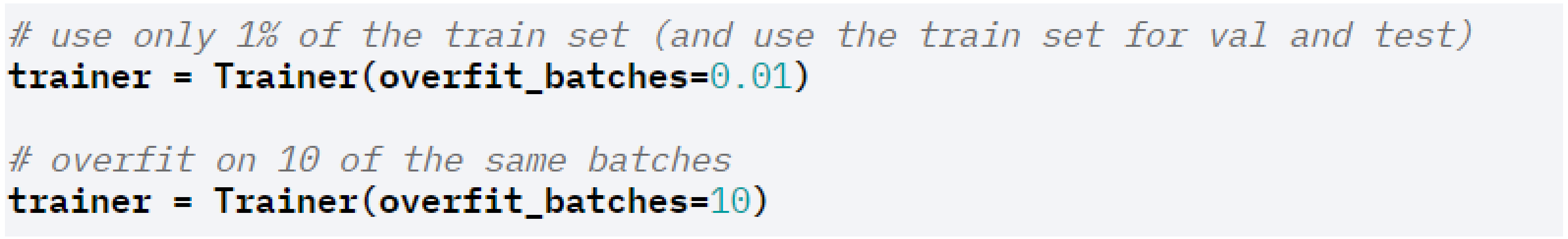


```
trainer = Trainer(gpus=4, log_gpu_memory='min_max')
```

- Settings are ‘all’ to record all GPUs or ‘min_max’ to record only min and max performing GPUs (less logging)

Overfit Batches

- Used to train the model on a single batch without shuffling to overfit and make sure that the architecture is working and there are no logic bugs.
- Input can be a floating point or an integer.

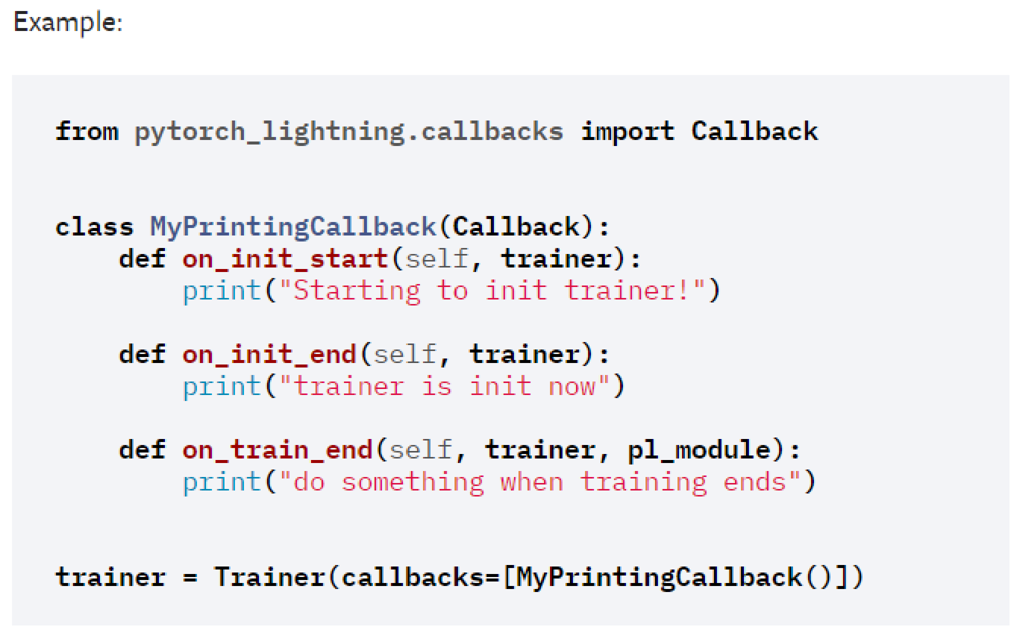


```python
# use only 1% of the train set (and use the train set for val and test)
trainer = Trainer(overfit_batches=0.01)

# overfit on 10 of the same batches
trainer = Trainer(overfit_batches=10)
```

- Floating points represent a % of the entire training set and integers represent the number of batches to use.

Callbacks

- The trainer also has a callback parameter where an array of callback functions can be passed in with hooks that cause executions of those callbacks at different times.

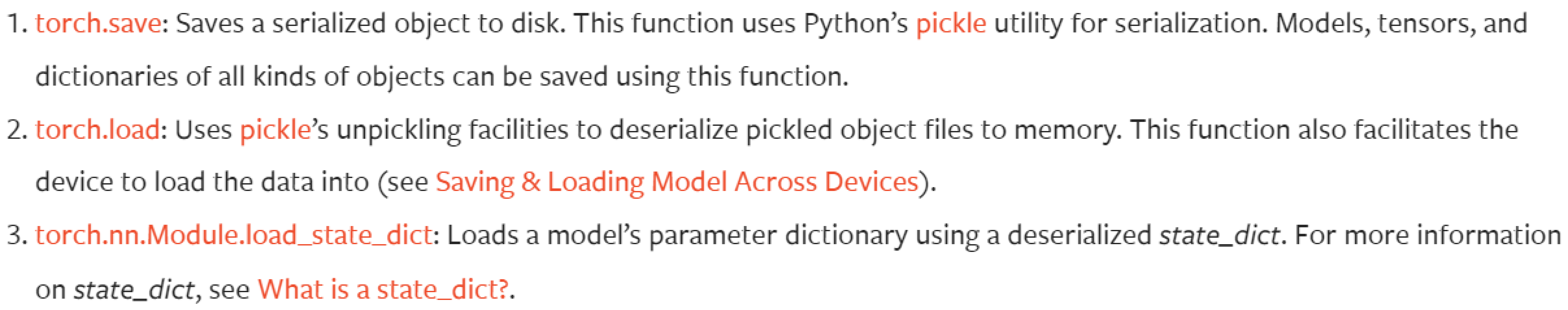


Example:

```
from pytorch_lightning.callbacks import Callback


class MyPrintingCallback(Callback):
    def on_init_start(self, trainer):
        print("Starting to init trainer!")

    def on_init_end(self, trainer):
        print("trainer is init now")

    def on_train_end(self, trainer, pl_module):
        print("do something when training ends")


trainer = Trainer(callbacks=[MyPrintingCallback()])
```

Saving and Loading Model

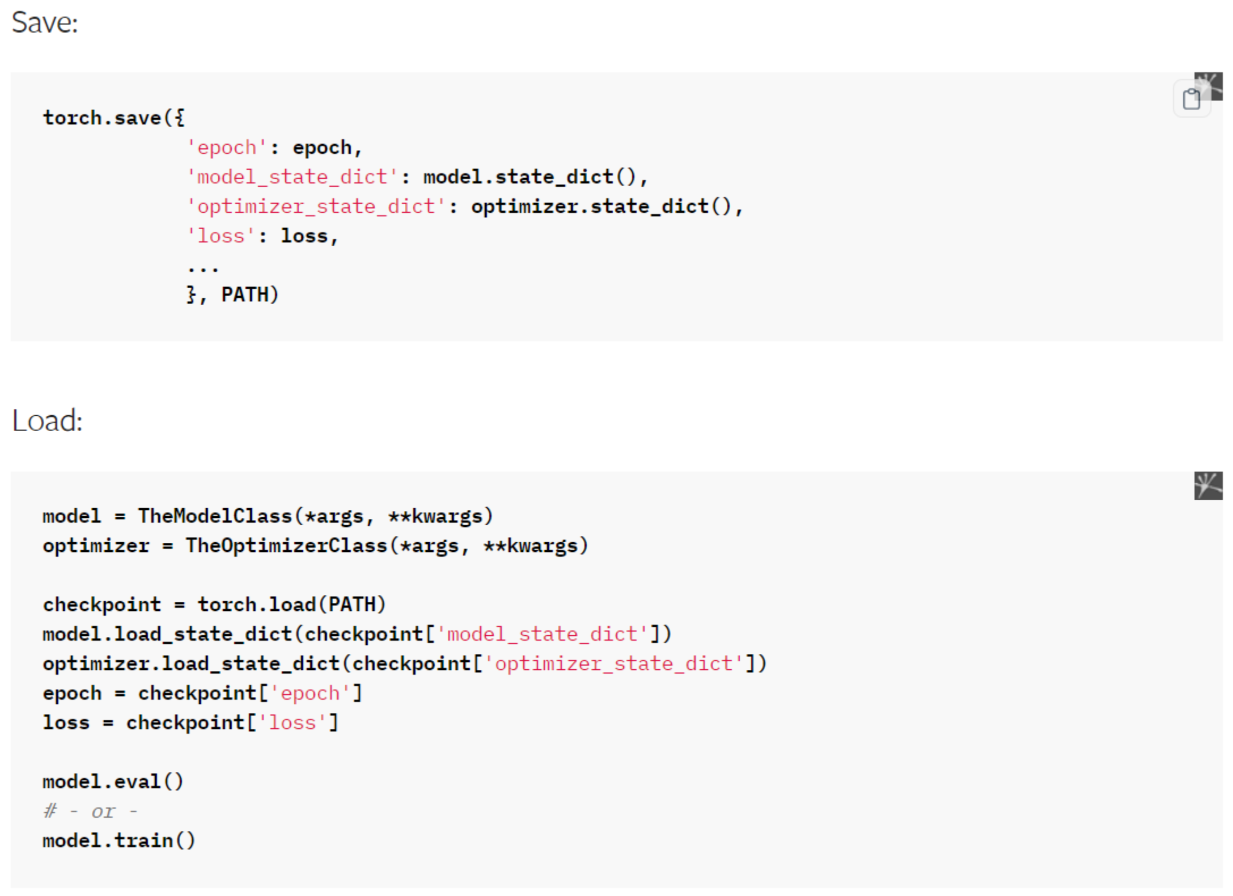
- A *state_dict* is simply a Python dictionary object that maps each layer to its parameter tensor.



1. torch.save: Saves a serialized object to disk. This function uses Python's pickle utility for serialization. Models, tensors, and dictionaries of all kinds of objects can be saved using this function.
2. torch.load: Uses pickle's unpickling facilities to deserialize pickled object files to memory. This function also facilitates the device to load the data into (see Saving & Loading Model Across Devices).
3. torch.nn.Module.load_state_dict: Loads a model's parameter dictionary using a deserialized *state_dict*. For more information on *state_dict*, see What is a state_dict?.

Pytorch save and load model

- This is the conventional way of saving and loading models with pyTorch, but the LightningModule has automated checkpoints and included save/load support.



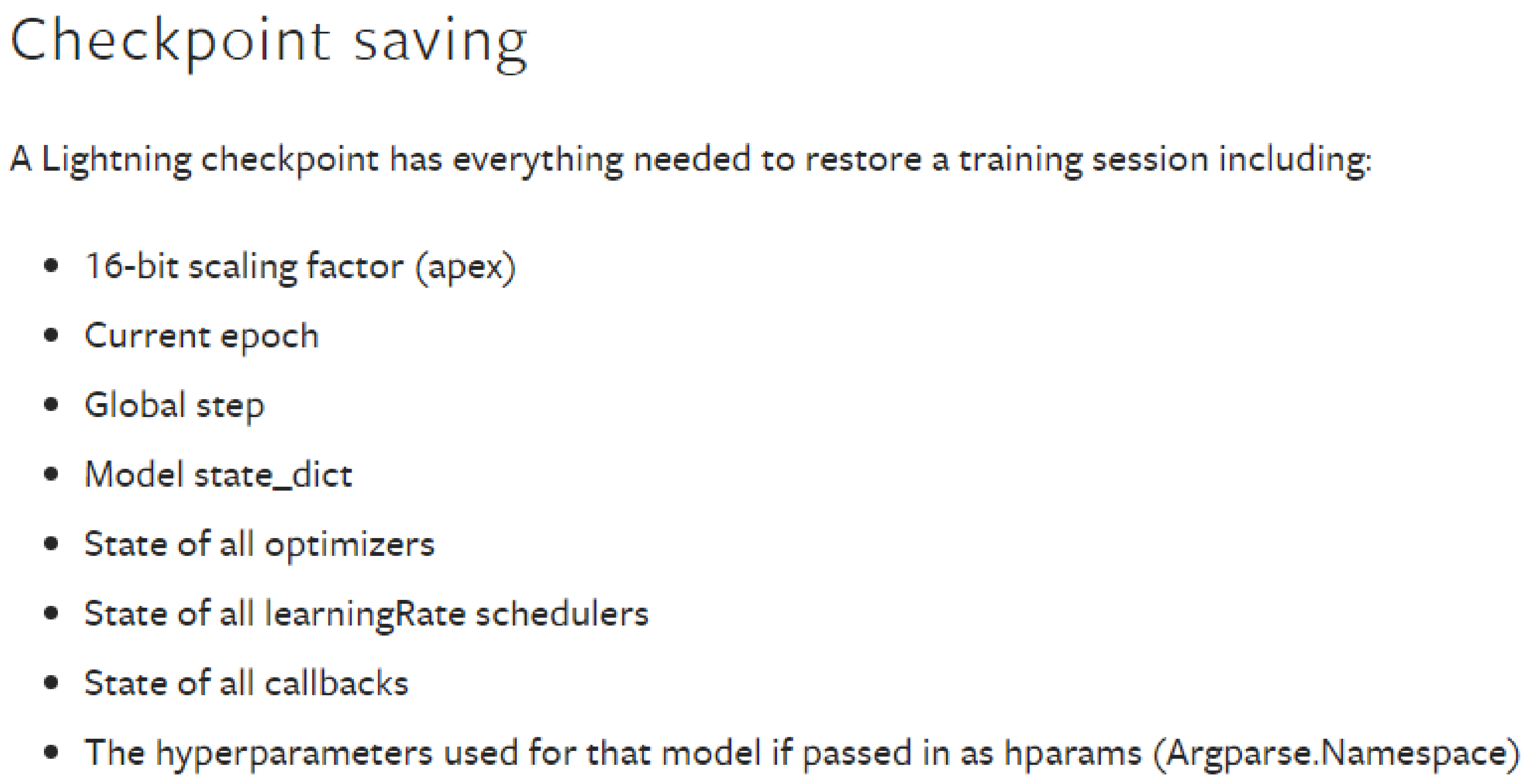
Save:

```
torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss,
            ...
            }, PATH)
```

Load:

```
model = TheModelClass(*args, **kwargs)
optimizer = TheOptimizerClass(*args, **kwargs)

checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']

model.eval()
# - or -
model.train()
```

Saving and Loading Weights with LightningModule



Checkpoint saving

A Lightning checkpoint has everything needed to restore a training session including:

- 16-bit scaling factor (apex)
- Current epoch
- Global step
- Model state_dict
- State of all optimizers
- State of all learningRate schedulers
- State of all callbacks
- The hyperparameters used for that model if passed in as hparams (Argparse.Namespace)

Saving/Loading Training

ModelCheckpoint (callback)

- Saves the model weights based on epochs, time intervals, or batches.
- The directory where the checkpoint must be saves is passed in and it automatically saves it there.
- In the implementation a period of 1 is used which corresponds to one save per epoch.

Resume_from_checkpoint (Trainer parameter)

- Resumes training from a ModelCheckpoint save.

Resources for Saving and Loading

- Model Checkpoint

https://pytorch-lightning.readthedocs.io/en/stable/api/pytorch_lightning.callbacks.model_checkpoint.html

- Saving files to the cloud with WandB

https://docs.wandb.ai/guides/track/advanced/save-restore

Notable Hooks for Trainer Callbacks

- on_train_batch_end
- on_train_epoch_end
- on_validation_epoch_end
- on_test_epoch_end
- on_keyboard_interrupt
- on_exception

Notable ModelCheckpoint parameters

Inputs

- train_time_interval
- every_n_train_steps
- every_n_epochs

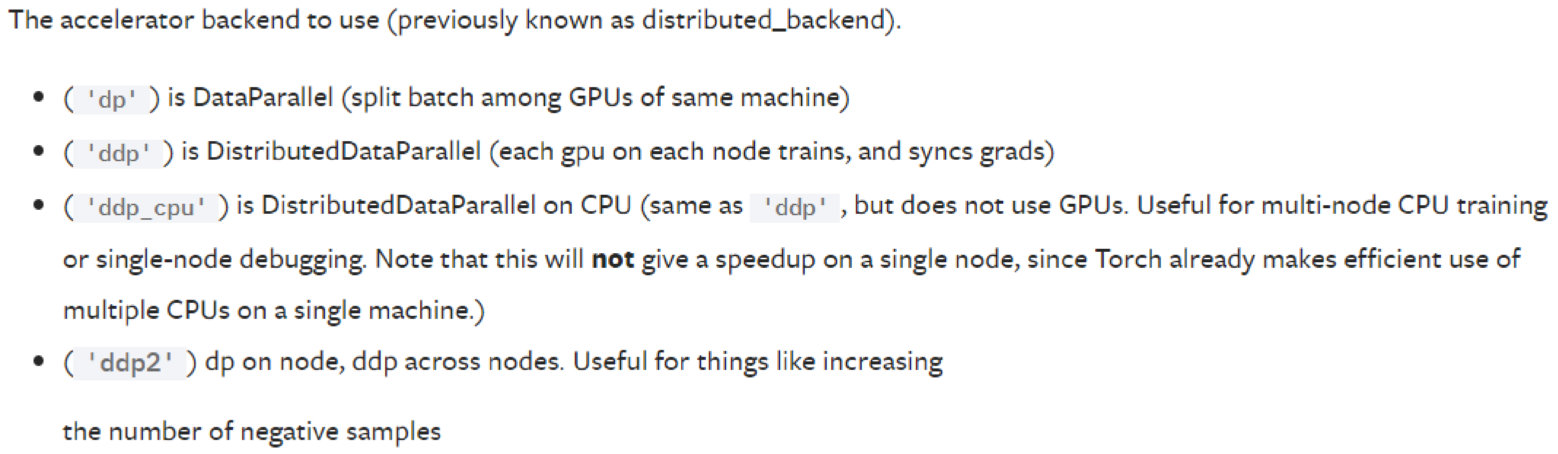
Hooks

- on_validation_end

Accelerators

- They are methods used to speed up training. Implementation currently uses DDP.
- Distributed Data Parallel
- Every GPU across every machine gets a copy of the model and a subset of the data. The GPU will only see that subset. The model will train a forward and backward pass locally on each GPU and then the weights are synced globally to pass on to optimizers which update all GPUs.
- Each GPU instantiates its own copy of the model so the same seed must be used such that weights don’t change.

https://www.youtube.com/watch?v=55fHcXNBkEY



The accelerator backend to use (previously known as distributed_backend).

- (`'dp'`) is DataParallel (split batch among GPUs of same machine)
- (`'ddp'`) is DistributedDataParallel (each gpu on each node trains, and syncs grads)
- (`'ddp_cpu'`) is DistributedDataParallel on CPU (same as `'ddp'`, but does not use GPUs. Useful for multi-node CPU training or single-node debugging. Note that this will **not** give a speedup on a single node, since Torch already makes efficient use of multiple CPUs on a single machine.)
- (`'ddp2'`) dp on node, ddp across nodes. Useful for things like increasing

  the number of negative samples

Running

1. Run with fast_dev_run on (Quick Debugging check)
2. Run one epoch with log_gpu_memory on to verify that the parameters are optimizing gpu use maximally
3. Run model with overfit_batches on to verify that training works and there are no logic errors in the model
4. Train the model, filters will be saved at the end of every epoch and logs are made every batch.

Scheduler

- Reduce on Plateau

https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html#torch.optim.lr_scheduler.ReduceLROnPlateau

Pytorch Available Multiclass Model Metrics

Highlighted = Used

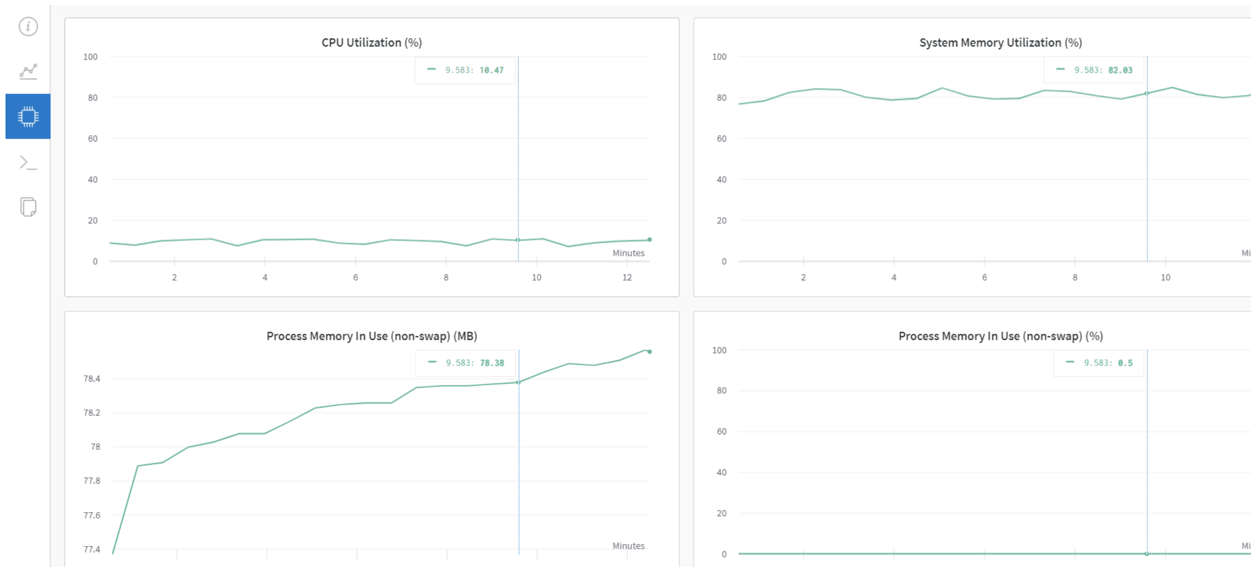
- Accuracy – Top k Accuracy
- AveragePrecision - summarises the precision recall curve into one number
- ROC - Computes Receiver Operating Characteristic: TPR vs. FPR
- AUROC - Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC). Multiclass uses one vs rest.
- Confusion Matrix – TP vs. FP vs TN vs FN
- IoU – Intersection Over Union (Bouding Boxes)

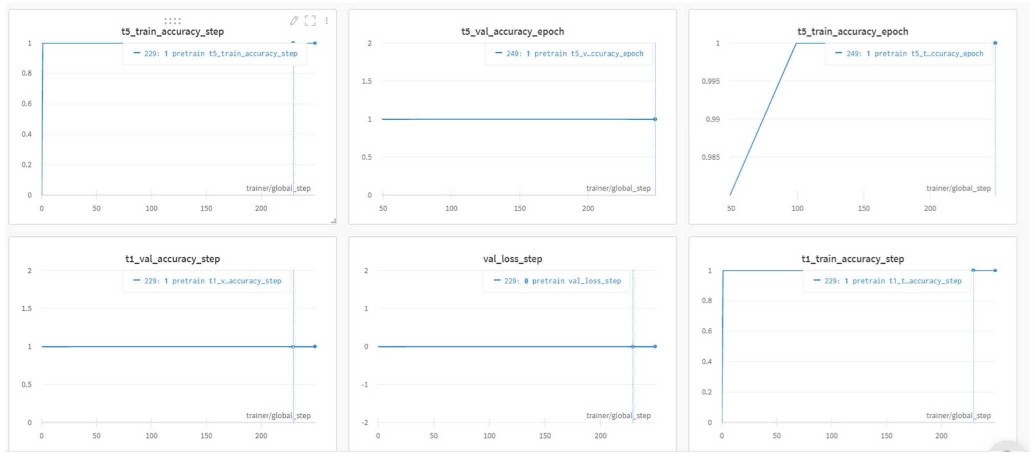
Many Many Many others

https://torchmetrics.readthedocs.io/en/latest/references/modules.html#id3

Logging

- Wandb (Weights and biases) logger is used
- Provides online interface for data logged, machine performance data, config, etc.


CPU Utilization (%)
System Memory Utilization (%)
Process Memory In Use (non-swap) (MB)
Process Memory In Use (non-swap) (%)
Minutes



Visualization

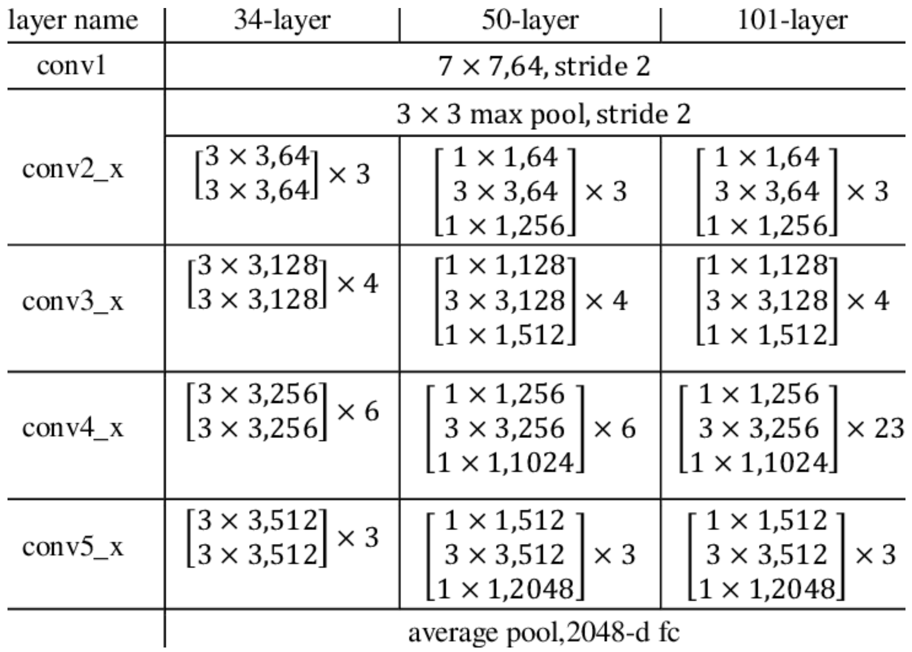
- A visualization approach to understand what the model is perceiving is plotting the convolutional filters. This is implemented in the plotFilters() function, there are several available ways to represent a color gradient and print these, the main ones are grey scale and gmap scale (green tones).
- Feature map visualizations are also done by saving each one of the model's layer in an array, running them individually and visualizing each two-dimensional layer in each of the tensors as an image.

Note: We should also try occlusion which is greying out parts of the input to see which features the model is the most dependent on to obtain a right prediction.

*Class level filter visualizations can be done

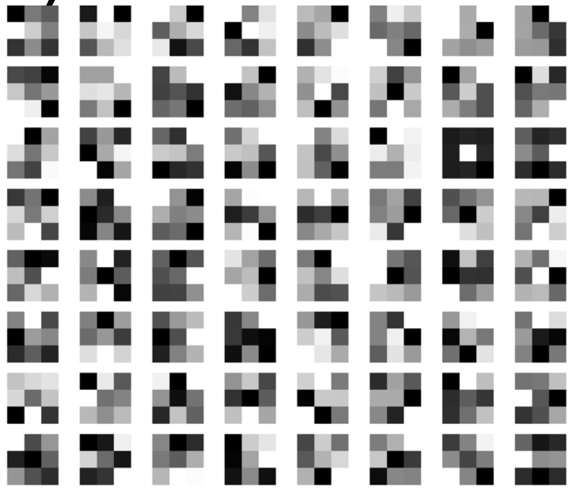
Filters Visualization

- Resnet visualized filters match the architecture.
- Blacked out filters belong to 1x1 convolutions (Dimensionality change).

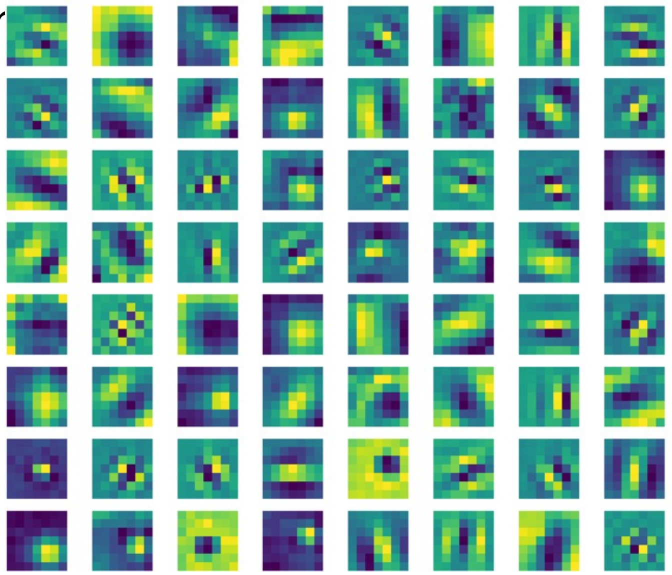


layer name	34-layer	50-layer	101-layer
conv1	$7 \times 7, 64$, stride 2		
conv2_x	3×3 max pool, stride 2		
	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$
conv4_x	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$
conv5_x	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	average pool, 2048-d fc		

Resnet Architecture
(Ir



Grey Scale layer 3 filters



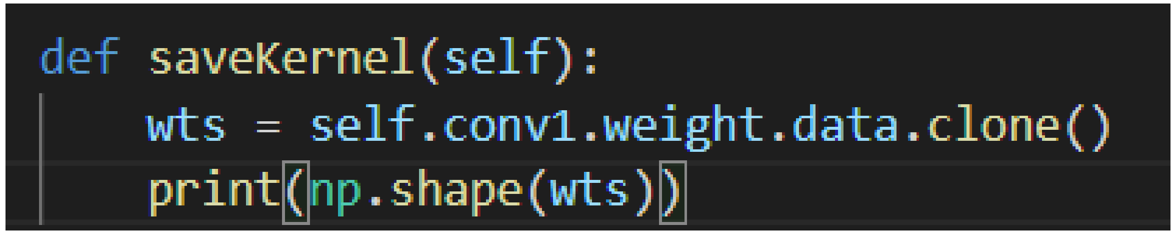
GMAP layer 1 filters

Filters Visualization Implementation

- The plotFilters() function is used which takes in a pytorch model and iterates over its layers printing each filter (top down approach).
- plotFilters() uses get_plot_dimensions to find the dimensions to give the plot with a given number of filters.
- Visualization is implemented by passing in a callback (a class called visualize) to the lightning Trainer class under the “callbacks” input parameter.
- The hook used in the visualize callback class is on_train_epoch_start.
- This class also takes in a path to save the filters and keeps a counter of the epoch.

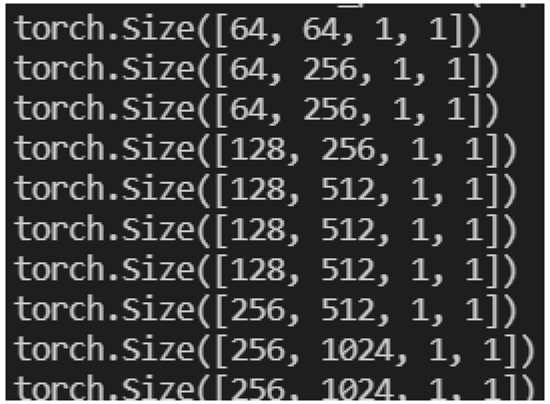
Class based “Bottom Up” filter visualization

- In the case of using class based visualization the following formula would be function would be used to obtain the weights from a specific layer.



```python
def saveKernel(self):
    wts = self.conv1.weight.data.clone()
    print(np.shape(wts))
```

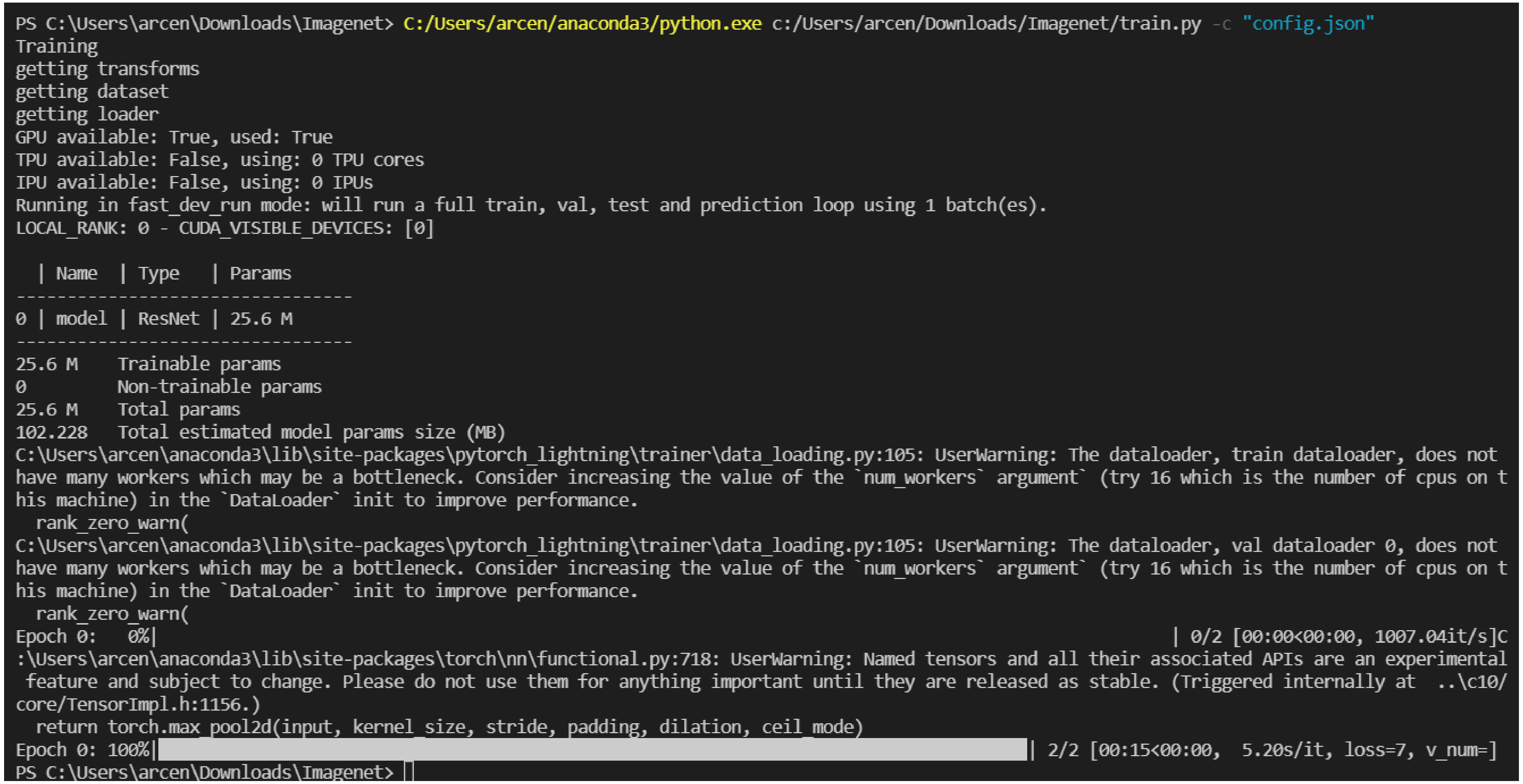
Function used, conv1 is a convolutional layer and its weights are extracted.



```
torch.Size([64, 64, 1, 1])
torch.Size([64, 256, 1, 1])
torch.Size([64, 256, 1, 1])
torch.Size([128, 256, 1, 1])
torch.Size([128, 512, 1, 1])
torch.Size([128, 512, 1, 1])
torch.Size([128, 512, 1, 1])
torch.Size([256, 512, 1, 1])
torch.Size([256, 1024, 1, 1])
torch.Size([256, 1024, 1, 1])
```

Dimensions of object printed in the function on the left.

Fast Dev Run



```
PS C:\Users\arcen\Downloads\Imagenet> C:/Users/arcen/anaconda3/python.exe c:/Users/arcen/Downloads/Imagenet/train.py -c "config.json"
Training
getting transforms
getting dataset
getting loader
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
Running in fast_dev_run mode: will run a full train, val, test and prediction loop using 1 batch(es).
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name  | Type   | Params
---------------------------------
0 | model | ResNet | 25.6 M
---------------------------------
25.6 M    Trainable params
0         Non-trainable params
25.6 M    Total params
102.228   Total estimated model params size (MB)
C:\Users\arcen\anaconda3\lib\site-packages\pytorch_lightning\trainer\data_loading.py:105: UserWarning: The dataloader, train dataloader, does not
have many workers which may be a bottleneck. Consider increasing the value of the `num_workers` argument` (try 16 which is the number of cpus on t
his machine) in the `DataLoader` init to improve performance.
  rank_zero_warn(
C:\Users\arcen\anaconda3\lib\site-packages\pytorch_lightning\trainer\data_loading.py:105: UserWarning: The dataloader, val dataloader 0, does not
have many workers which may be a bottleneck. Consider increasing the value of the `num_workers` argument` (try 16 which is the number of cpus on t
his machine) in the `DataLoader` init to improve performance.
  rank_zero_warn(
Epoch 0:   0%|                                                                                         | 0/2 [00:00<00:00, 1007.04it/s]C
:\Users\arcen\anaconda3\lib\site-packages\torch\nn\functional.py:718: UserWarning: Named tensors and all their associated APIs are an experimental
 feature and subject to change. Please do not use them for anything important until they are released as stable. (Triggered internally at  ..\c10/
core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
Epoch 0: 100%|█████████████████████████████████████████████████████████████████████| 2/2 [00:15<00:00,  5.20s/it, loss=7, v_num=]
PS C:\Users\arcen\Downloads\Imagenet> 
```

Parameters We can Evaluate

- Transforms
- Loss function
- Optimizer
- Scheduler
- GPU optimization
- Trainer Params Precision?

Config parameters we can incorporate

- Resnet models
- Model in general (https://github.com/MrtnMndt/Deep_Openset_Recognition_through_Uncertainty)
- Transforms
- Datasets

Confused Logits

- https://pytorch-lightning-bolts.readthedocs.io/en/0.2.0/api/pl_bolts.callbacks.vision.confused_logit.html

Callback function that gets activated when two class probabilities are similar. This changes the input in the direction that causes the model to chose one class over the other.

This looks very promising to visualize what features the model considers to be prominent in one class versus another.

Git issue

- View files in current commit

https://stackoverflow.com/questions/8533202/list-files-in-local-git-repo

- Delete all commit history (fix)

https://stackoverflow.com/questions/13716658/how-to-delete-all-commit-history-in-github

FOTS

Base Model

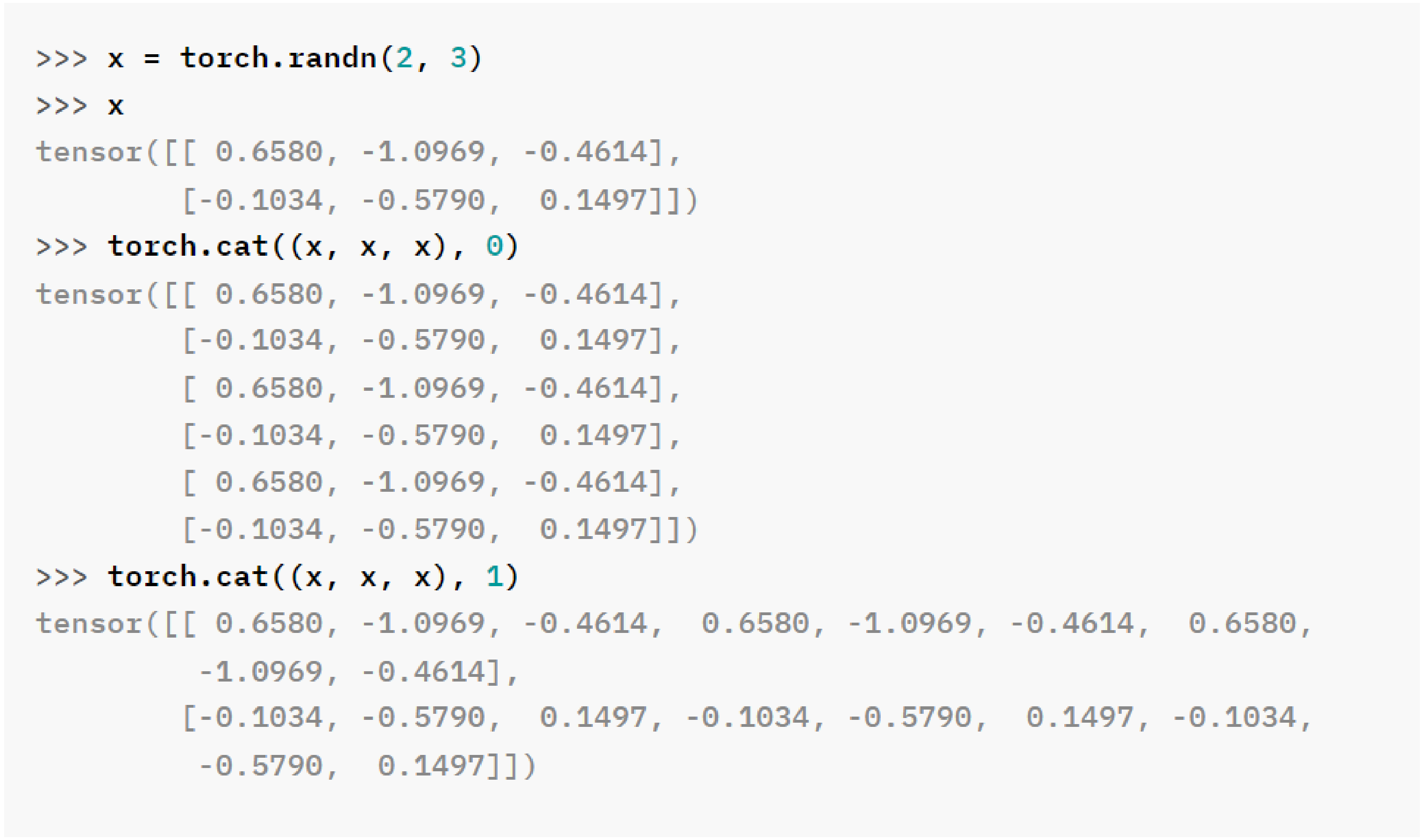
Base class for all models used to log a summary of the parametrizable weights in the model.

Concatenation Torch.cat

Concatenates the input tensors in the specified dimension.

Dimension 0 means one is placed after the other (like array concat).

Dimension 1 means each of their elements are concatenated. (Element 1 with 1, 2 with 2 etc.)



```
>>> x = torch.randn(2, 3)
>>> x
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 0)
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 1)
tensor([[ 0.6580, -1.0969, -0.4614,  0.6580, -1.0969, -0.4614,  0.6580,
         -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497, -0.1034, -0.5790,  0.1497, -0.1034,
         -0.5790,  0.1497]])
```

Torch.Split

- Does the inverse operation of torch.cat()

https://pytorch.org/docs/stable/generated/torch.split.html

Shared Conv

- Inputs: Backbone model and config file

Forward:

1. Iterates through each layer in the backbone saving the output
2. Runs the tensor through a dummy layer (no effect) and “unpools” it (upsampling) by using interpolation.
3. Runs concatenation (except for the first layer) hidden layers followed by unpooling three times, the last unpooling is omitted.
    - Each layer takes the last layer’s output and the output of a layer from resnet, concatenates them and runs the hidden layer convolutions.
4. Runs a final 3x3 convolutional layer with a padding of 1
5. Batch normalization followed by reLU

Output = 32 channels

Hidden Layers

- Takes in two tensors and concatenates them in the first dimension (element wise, each element in one tensor is concatenated with each element in the other to form the new tensor)
- Runs 1x1 convolution to reduce the channels.
- Batch normalization with momentum set to 0.003
- 3x3 convolution with padding = 1
- Final Batch Normalizaiton with momentum set to 0.003

Shared Conv

- In the github implementation they have two convolutions instead of one, one is used to reduce feature channels (1x1 filter) but then there's another 3x3 filter convolution.

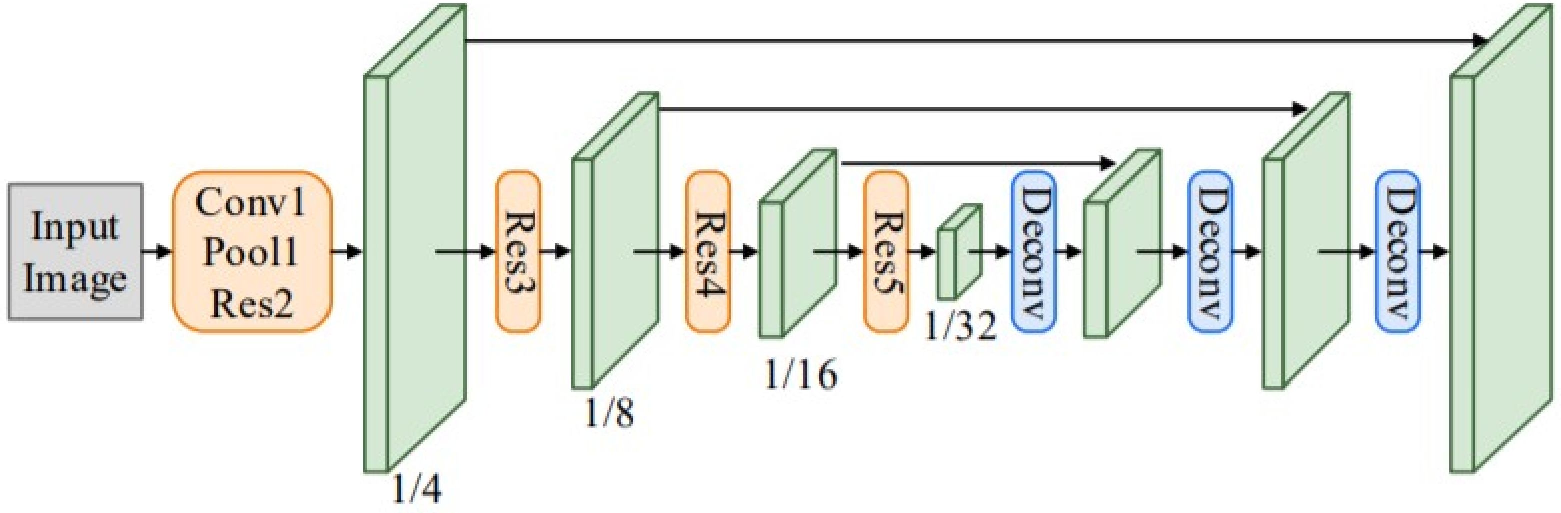


Figure 3: Architecture of shared convolutions. Conv1-Res5 are operations from ResNet-50, and Deconv consists of one convolution to reduce feature channels and one bilinear upsampling operation.

Interpolation – pytorch function

- Upsamples the tensor using interpolation methods.
- FOTS Implementation:

Uses bilinear interpolation (Linear interpolation in one direction and then in another, [2d])

Scales the tensor to be twice the size

Uses the align corners flag, which aligns the outputs tensors corner and center to the input tensor and expands in between.

The alternative (false) aligns the the corners and expand the edges going outside the boundaries (padding)

Detector

- Input: Feature map created by shared conv
- Output: score map and geometry

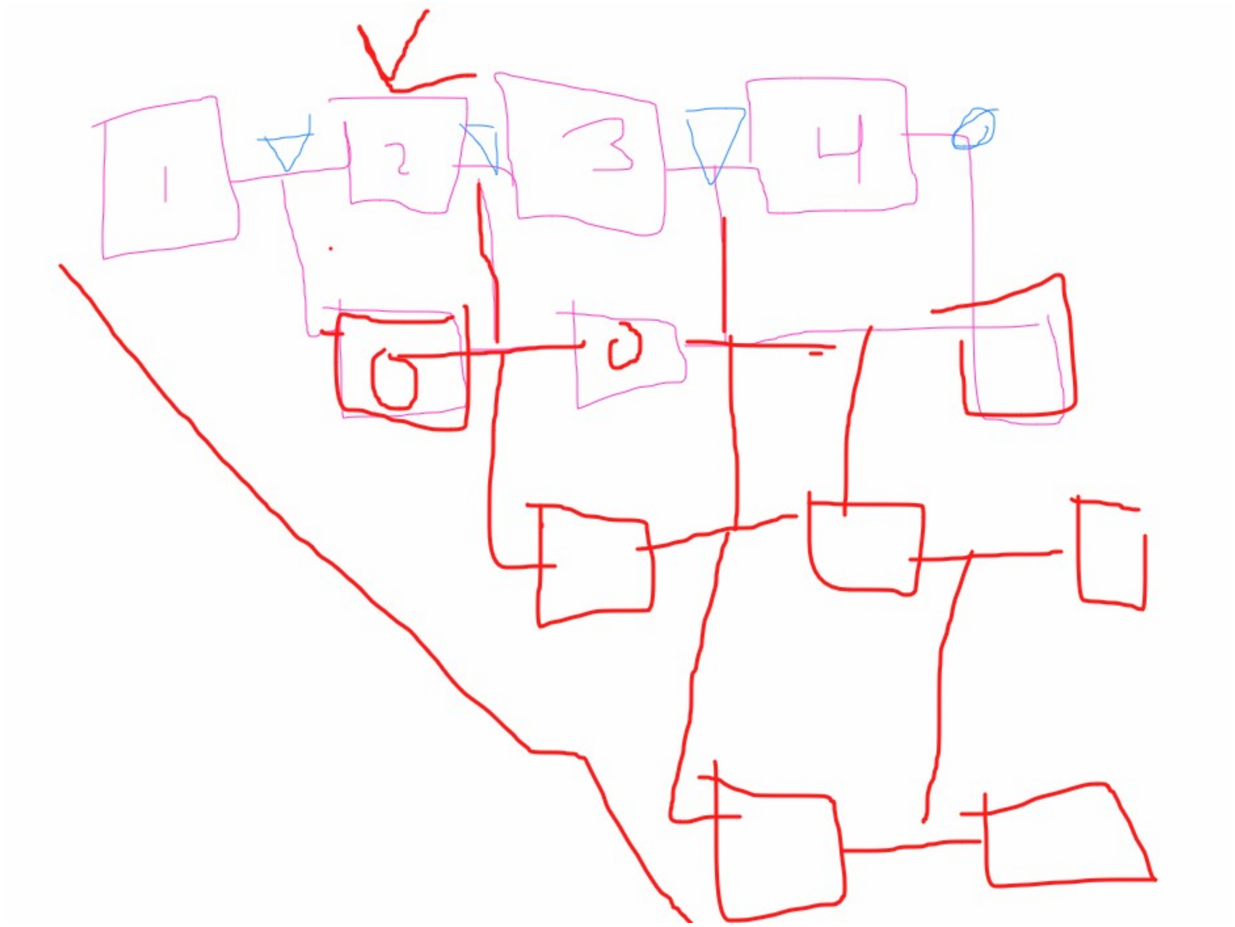
The **score map** is made by passing the input through a conv layer that reduces it to one channel and passing that through a sigmoid.

The **geometry** is a concatenation in the first dimension (element wise; each element in the tensor is concatenated) of the geomap and angle map.

The **geomap** is made by passing the input through a conv layer that reduces it to 4 channels and passing that through a sigmoid, the output is then multiplied by the image size.

The **angle map** is made by passing the input through a conv layer that reduces it to one channel and passing that through a sigmoid, subtracting 0.5 from each component and multiplying them by pi.

Network Parallelization Idea



Numpy Functions Used for pixel predictions

Argwhere, returns indexes in array with given condition

- https://numpy.org/doc/stable/reference/generated/numpy.argwhere.html

Argsort, returns indexes with orders of values in array

- https://numpy.org/doc/stable/reference/generated/numpy.argsort.html

Indexing

- https://numpy.org/doc/stable/reference/arrays.indexing.html

GPU Optimization

Mini Batches

- Larger batches means less generalization, the optimal number depends on the model but several papers agree on 32 being best.

https://stats.stackexchange.com/questions/164876/what-is-the-trade-off-between-batch-size-and-number-of-iterations-to-train-a-neu

https://wandb.ai/ayush-thakur/dl-question-bank/reports/What-s-the-Optimal-Batch-Size-to-Train-a-Neural-Network---VmlldzoyMDkyNDU

- Calculating the memory needed to train a specific model

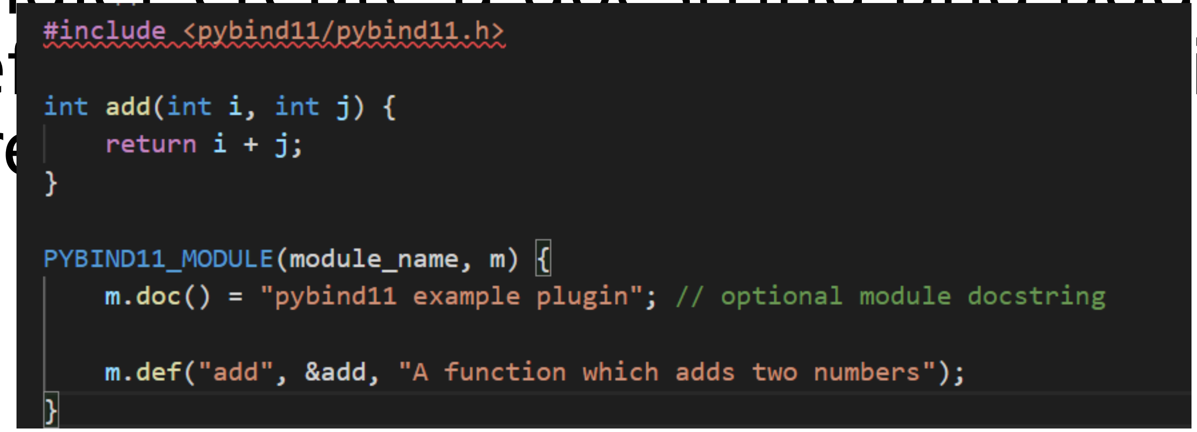
https://datascience.stackexchange.com/questions/12649/how-to-calculate-the-mini-batch-memory-impact-when-tr

How pytorch handles multiple GPUs

- https://www.youtube.com/watch?v=a6_pY9WwqdQ

Compiling C++ Code In python

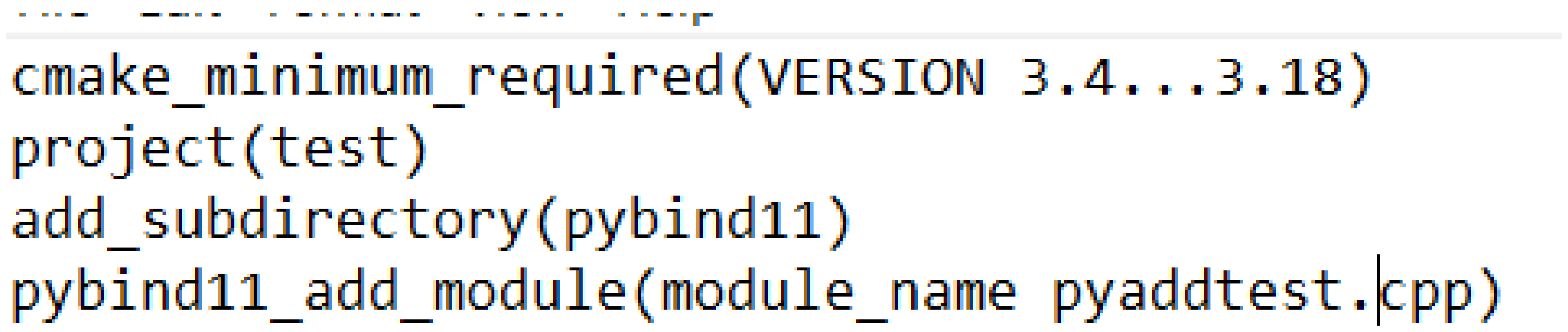
- Pybind11 allows compiling C++ code as a python module, callable from any script.
- Create your c file, import pybind and its headerfile and write the function.
- Call the pybind module macro, give it a name and a handler.
- Using the handler create a docstring and add the functions as handler.def ing them the c function addre



```cpp
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(module_name, m) {
    m.doc() = "pybind11 example plugin"; // optional module docstring

    m.def("add", &add, "A function which adds two numbers");
}
```

Cmake File

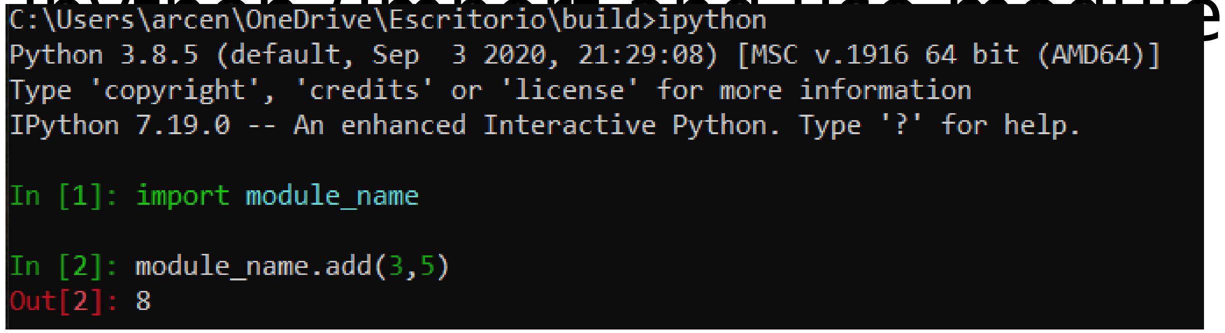
- Specify required version Required
- Give the project a name
- Add pybind directory cloned github)
- Add pybind module



```
cmake_minimum_required(VERSION 3.4...3.18)
project(test)
add_subdirectory(pybind11)
pybind11_add_module(module_name pyaddtest.cpp)
```

Running Cmake

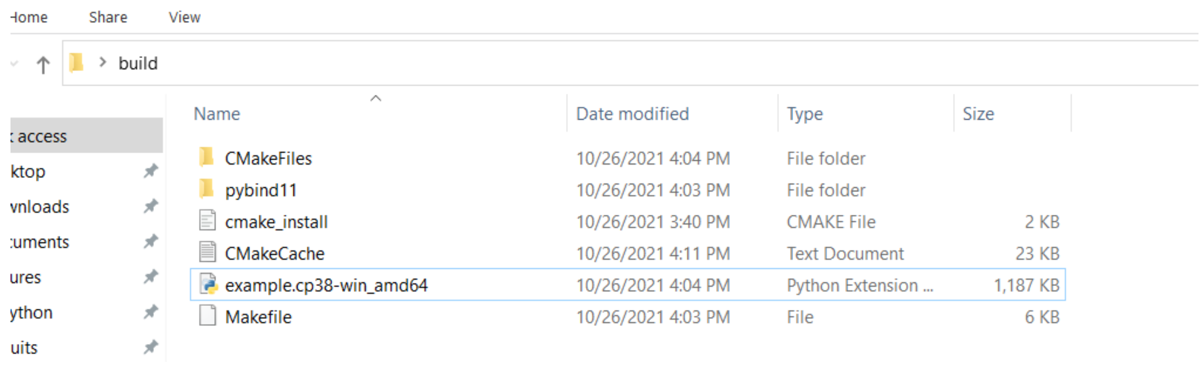
- Create a build directory and cd into it
- Set the generator with cmake .. –G “generator_name” or run configure on the GUI and select it
- The generator is the compiler, MinGW (“MinGW Makefiles”) can be used or Visual Studio (Not visual studio code).
- Run make on installed compiler: mingw32-make
- Test with ipython (import and use module).



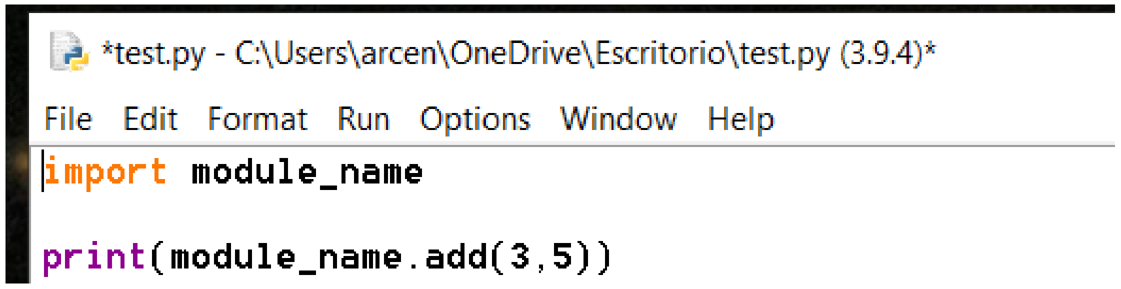
```
C:\Users\arcen\OneDrive\Escritorio\build>ipython
Python 3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import module_name

In [2]: module_name.add(3,5)
Out[2]: 8
```

Results

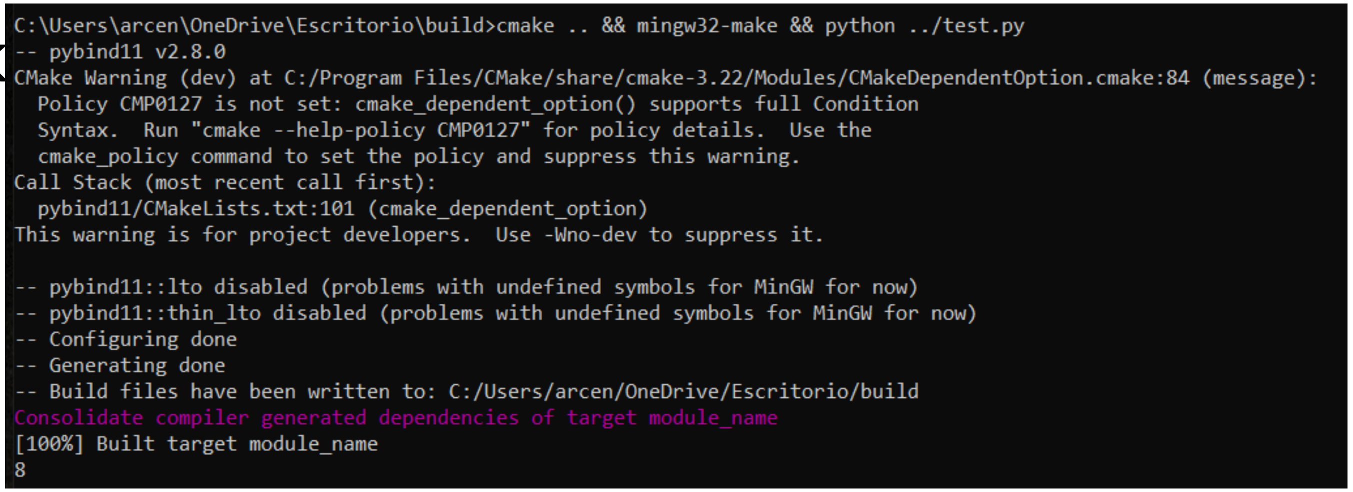
- A python extension file will be created with the module name.
- Module can be called like so in a python script.
- Script should be compiled as
- cmake .. && mingw32-mak && python ../filename.py
- Using mingw32 for make compiling.



Name	Date modified	Type	Size
CMakeFiles	10/26/2021 4:04 PM	File folder	
pybind11	10/26/2021 4:03 PM	File folder	
cmake_install	10/26/2021 3:40 PM	CMAKE File	2 KB
CMakeCache	10/26/2021 4:11 PM	Text Document	23 KB
example.cp38-win_amd64	10/26/2021 4:04 PM	Python Extension ...	1,187 KB
Makefile	10/26/2021 4:03 PM	File	6 KB


test.py - C:\Users\arcen\OneDrive\Escritorio\test.py (3.9.4)
File
Edit
Format
Run
Options
Window
Help

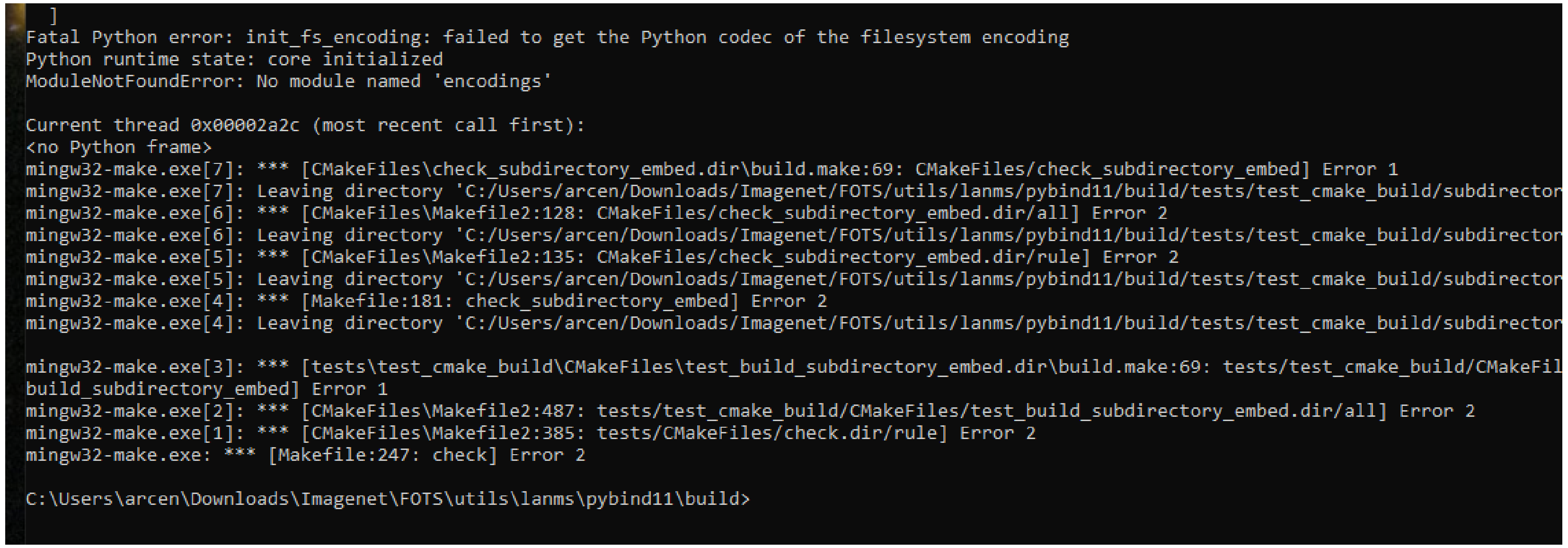
```python
import module_name

print(module_name.add(3,5))
```



```
C:\Users\arcen\OneDrive\Escritorio\build>cmake .. && mingw32-make && python ../test.py
-- pybind11 v2.8.0
CMake Warning (dev) at C:/Program Files/CMake/share/cmake-3.22/Modules/CMakeDependentOption.cmake:84 (message):
  Policy CMP0127 is not set: cmake_dependent_option() supports full Condition
  Syntax.  Run "cmake --help-policy CMP0127" for policy details.  Use the
  cmake_policy command to set the policy and suppress this warning.
Call Stack (most recent call first):
  pybind11/CMakeLists.txt:101 (cmake_dependent_option)
This warning is for project developers.  Use -Wno-dev to suppress it.

-- pybind11::lto disabled (problems with undefined symbols for MinGW for now)
-- pybind11::thin_lto disabled (problems with undefined symbols for MinGW for now)
-- Configuring done
-- Generating done
-- Build files have been written to: C:/Users/arcen/OneDrive/Escritorio/build
Consolidate compiler generated dependencies of target module_name
[100%] Built target module_name
8
```

Test error

- When PYTHONPATH is set to the paths given by sys.path:

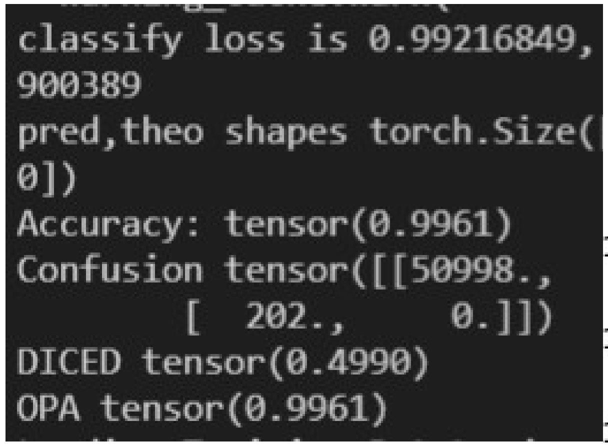


```
  ]
Fatal Python error: init_fs_encoding: failed to get the Python codec of the filesystem encoding
Python runtime state: core initialized
ModuleNotFoundError: No module named 'encodings'

Current thread 0x00002a2c (most recent call first):
<no Python frame>
mingw32-make.exe[7]: *** [CMakeFiles\check_subdirectory_embed.dir\build.make:69: CMakeFiles/check_subdirectory_embed] Error 1
mingw32-make.exe[7]: Leaving directory 'C:/Users/arcen/Downloads/Imagenet/FOTS/utils/lanms/pybind11/build/tests/test_cmake_build/subdirector
mingw32-make.exe[6]: *** [CMakeFiles\Makefile2:128: CMakeFiles/check_subdirectory_embed.dir/all] Error 2
mingw32-make.exe[6]: Leaving directory 'C:/Users/arcen/Downloads/Imagenet/FOTS/utils/lanms/pybind11/build/tests/test_cmake_build/subdirector
mingw32-make.exe[5]: *** [CMakeFiles\Makefile2:135: CMakeFiles/check_subdirectory_embed.dir/rule] Error 2
mingw32-make.exe[5]: Leaving directory 'C:/Users/arcen/Downloads/Imagenet/FOTS/utils/lanms/pybind11/build/tests/test_cmake_build/subdirector
mingw32-make.exe[4]: *** [Makefile:181: check_subdirectory_embed] Error 2
mingw32-make.exe[4]: Leaving directory 'C:/Users/arcen/Downloads/Imagenet/FOTS/utils/lanms/pybind11/build/tests/test_cmake_build/subdirector

mingw32-make.exe[3]: *** [tests\test_cmake_build\CMakeFiles\test_build_subdirectory_embed.dir\build.make:69: tests/test_cmake_build/CMakeFil
build_subdirectory_embed] Error 1
mingw32-make.exe[2]: *** [CMakeFiles\Makefile2:487: tests/test_cmake_build/CMakeFiles/test_build_subdirectory_embed.dir/all] Error 2
mingw32-make.exe[1]: *** [CMakeFiles\Makefile2:385: tests/CMakeFiles/check.dir/rule] Error 2
mingw32-make.exe: *** [Makefile:247: check] Error 2

C:\Users\arcen\Downloads\Imagenet\FOTS\utils\lanms\pybind11\build>
```

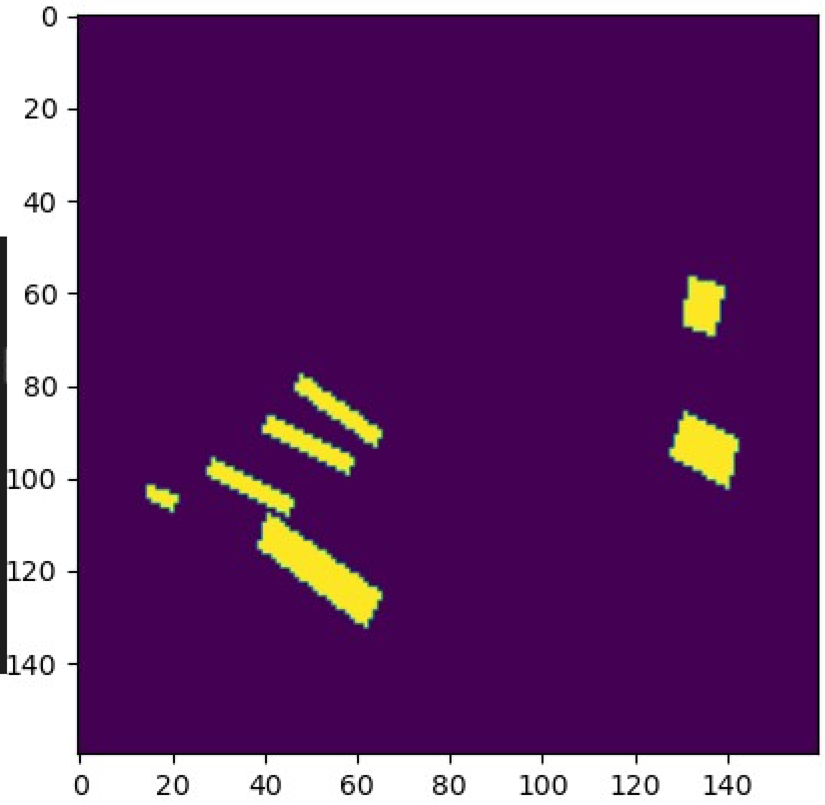
- This error is caused by having the wrong paths.

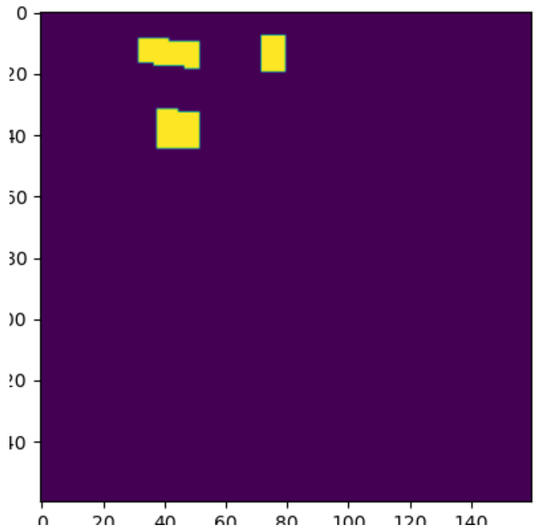
Metrics

Untrained

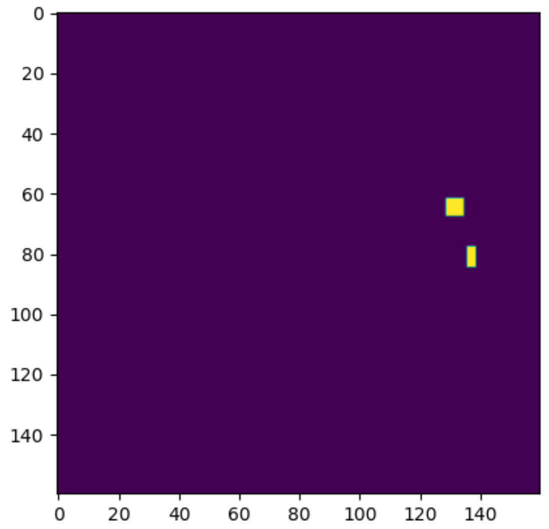


```
classify loss is 0.99216849,
900389
pred,theo shapes torch.Size(
0])
Accuracy: tensor(0.9961)
Confusion tensor([[50998.,
        [  202.,     0.]])
DICED tensor(0.4990)
OPA tensor(0.9961)
```



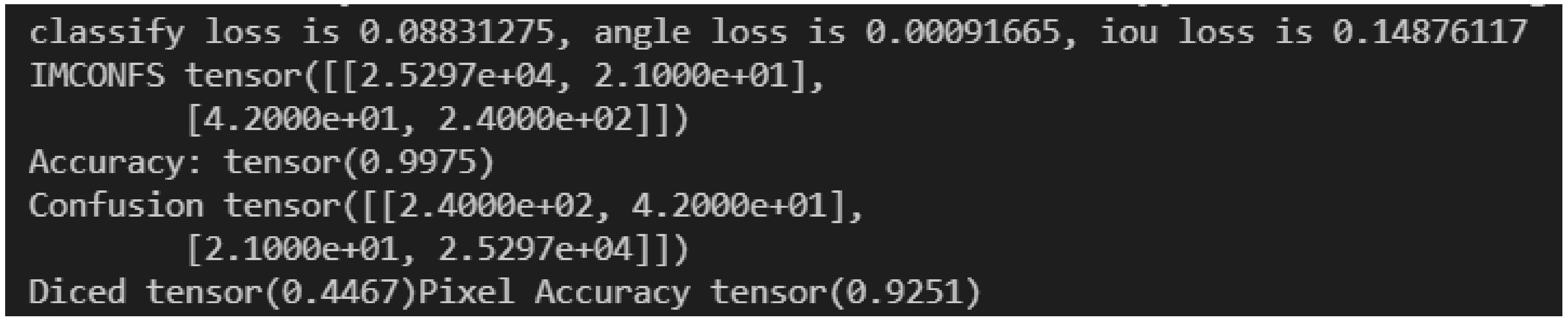


Predicted

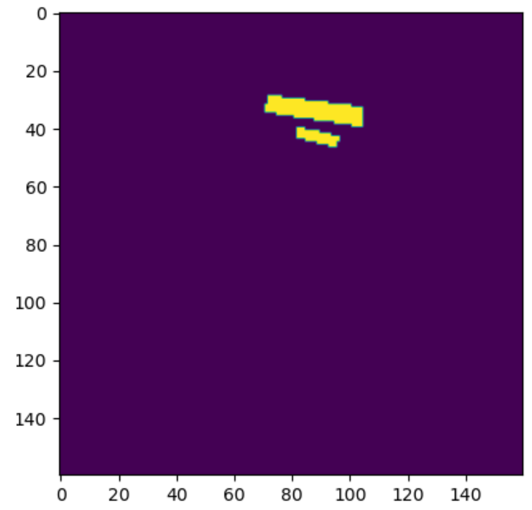


Theoretical

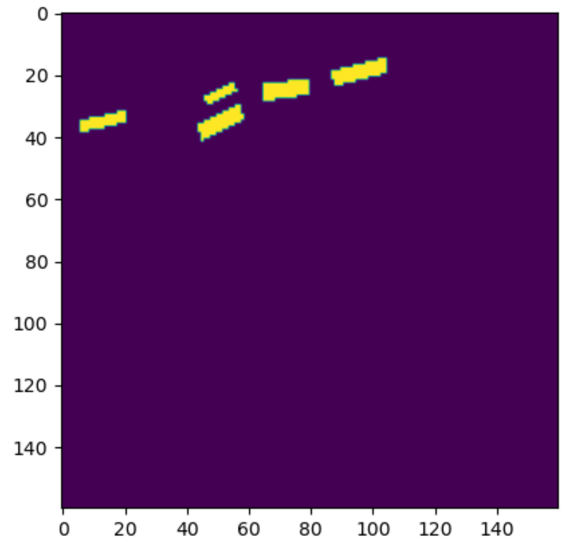
Trained



```
classify loss is 0.08831275, angle loss is 0.00091665, iou loss is 0.14876117
IMCONFS tensor([[2.5297e+04, 2.1000e+01],
        [4.2000e+01, 2.4000e+02]])
Accuracy: tensor(0.9975)
Confusion tensor([[2.4000e+02, 4.2000e+01],
        [2.1000e+01, 2.5297e+04]])
Diced tensor(0.4467)Pixel Accuracy tensor(0.9251)
```



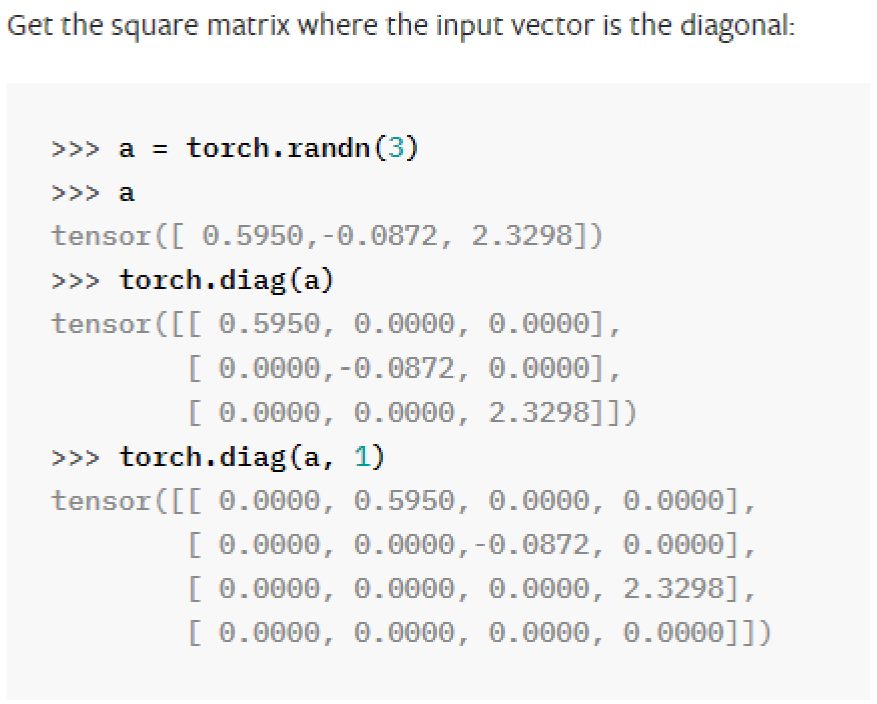
Predicted



Theoretical

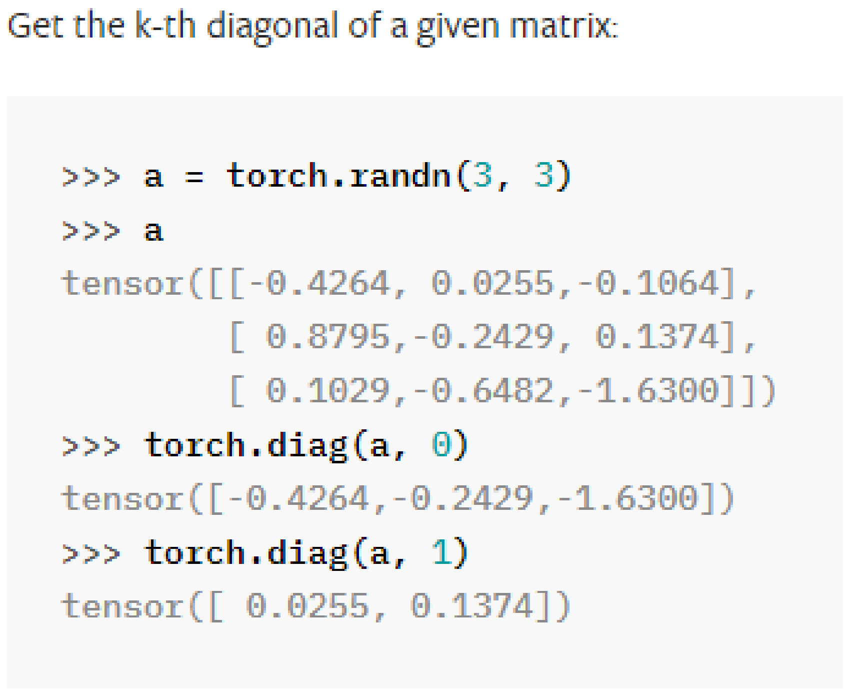
Torch.Diag

- Used to get



Get the square matrix where the input vector is the diagonal:

```
>>> a = torch.randn(3)
>>> a
tensor([ 0.5950,-0.0872, 2.3298])
>>> torch.diag(a)
tensor([[ 0.5950, 0.0000, 0.0000],
        [ 0.0000,-0.0872, 0.0000],
        [ 0.0000, 0.0000, 2.3298]])
>>> torch.diag(a, 1)
tensor([[ 0.0000, 0.5950, 0.0000, 0.0000],
        [ 0.0000, 0.0000,-0.0872, 0.0000],
        [ 0.0000, 0.0000, 0.0000, 2.3298],
        [ 0.0000, 0.0000, 0.0000, 0.0000]])
```



Get the k-th diagonal of a given matrix:

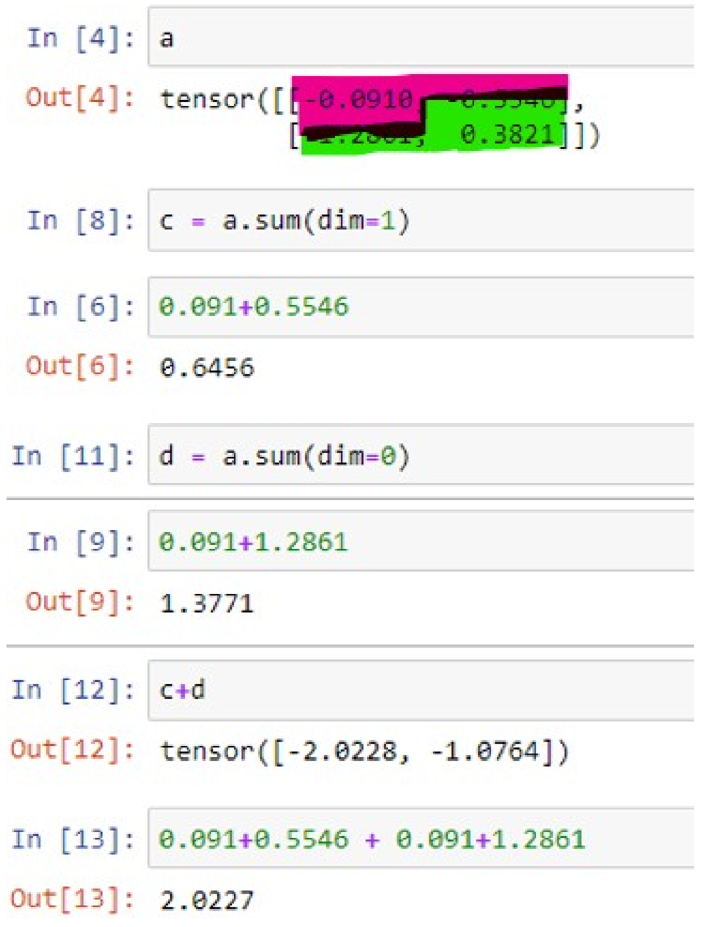
```
>>> a = torch.randn(3, 3)
>>> a
tensor([[-0.4264, 0.0255,-0.1064],
        [ 0.8795,-0.2429, 0.1374],
        [ 0.1029,-0.6482,-1.6300]])
>>> torch.diag(a, 0)
tensor([-0.4264,-0.2429,-1.6300])
>>> torch.diag(a, 1)
tensor([ 0.0255, 0.1374])
```

$$F_1\text{-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

Dice Coefficient

$$\text{Dice coefficient}(A, B) = \frac{2 \times |A \cap B|}{|A| + |B|}$$

- Intersection is obtained by adding all the correct predictions (diagonal of confusion matrix)
- The absolute value of a set is calculated as every prediction related to that set (true positives, false positives, and false negatives), it is calculated by using torch.sum() along both dimensions as shown in the notebook to the right.



```
In [4]:  a
Out[4]:  tensor([[-0.0910, -0.5546],
                  [-1.2861,  0.3821]])

In [8]:  c = a.sum(dim=1)

In [6]:  0.091+0.5546
Out[6]:  0.6456

In [11]: d = a.sum(dim=0)

In [9]:  0.091+1.2861
Out[9]:  1.3771

In [12]: c+d
Out[12]: tensor([-2.0228, -1.0764])

In [13]: 0.091+0.5546 + 0.091+1.2861
Out[13]: 2.0227
```

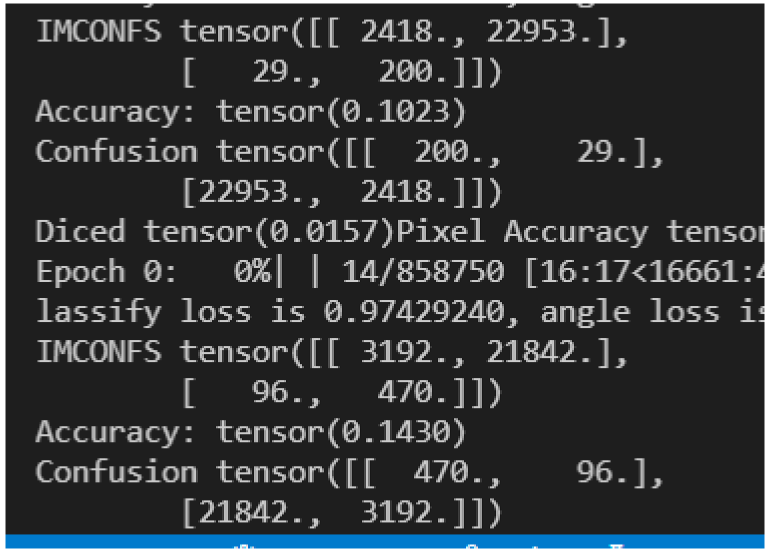
Colors show the summations done when computing the denominator of the dice coefficient with relation to the confusion matrix.

Change dice score?

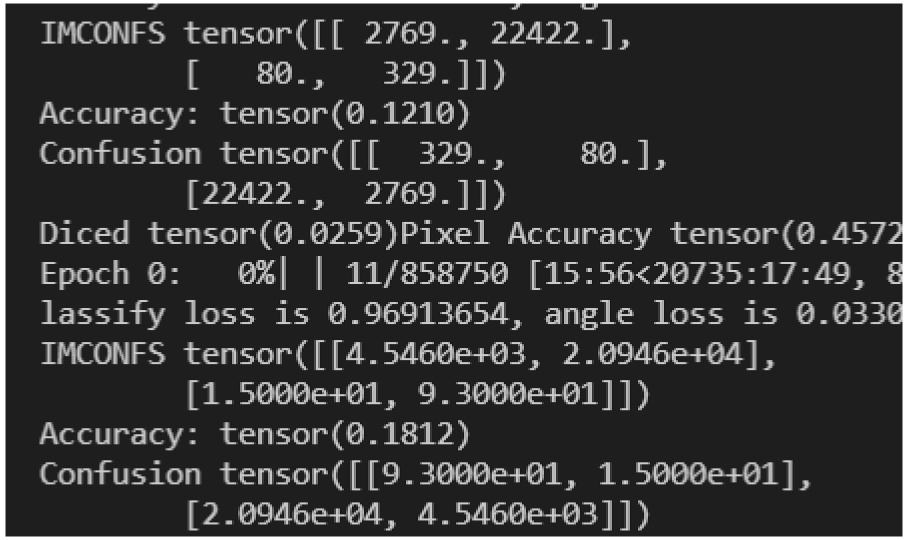
- Make it only true positives instead of true positives + true negatives
- This is used for image segmentation not detection (Multiclass)

Metrics

- No Training

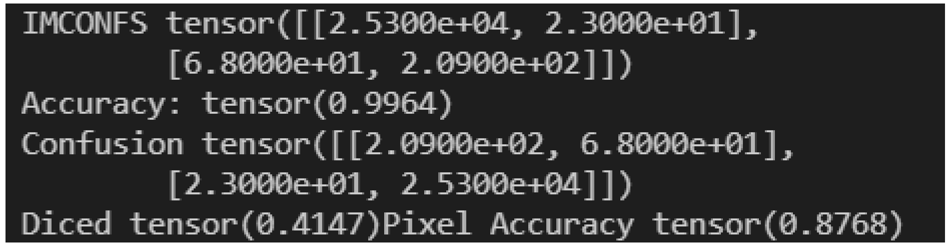


```
IMCONFS tensor([[ 2418., 22953.],
        [   29.,   200.]])
Accuracy: tensor(0.1023)
Confusion tensor([[  200.,    29.],
        [22953.,  2418.]])
Diced tensor(0.0157)Pixel Accuracy tenso
Epoch 0:   0%| | 14/858750 [16:17<16661:4
lassify loss is 0.97429240, angle loss i
IMCONFS tensor([[ 3192., 21842.],
        [   96.,   470.]])
Accuracy: tensor(0.1430)
Confusion tensor([[  470.,    96.],
        [21842.,  3192.]])
```

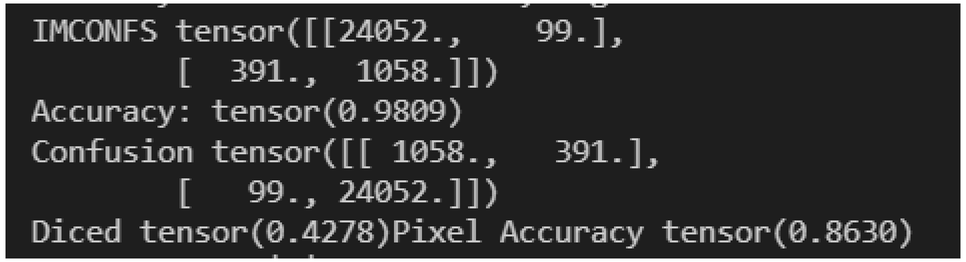


```
IMCONFS tensor([[ 2769., 22422.],
        [   80.,   329.]])
Accuracy: tensor(0.1210)
Confusion tensor([[  329.,    80.],
        [22422.,  2769.]])
Diced tensor(0.0259)Pixel Accuracy tensor(0.4572
Epoch 0:   0%| | 11/858750 [15:56<20735:17:49, 8
lassify loss is 0.96913654, angle loss is 0.0330
IMCONFS tensor([[4.5460e+03, 2.0946e+04],
        [1.5000e+01, 9.3000e+01]])
Accuracy: tensor(0.1812)
Confusion tensor([[9.3000e+01, 1.5000e+01],
        [2.0946e+04, 4.5460e+03]])
```

- 7 Epochs (interrupted)



```
IMCONFS tensor([[2.5300e+04, 2.3000e+01],
        [6.8000e+01, 2.0900e+02]])
Accuracy: tensor(0.9964)
Confusion tensor([[2.0900e+02, 6.8000e+01],
        [2.3000e+01, 2.5300e+04]])
Diced tensor(0.4147)Pixel Accuracy tensor(0.8768)
```



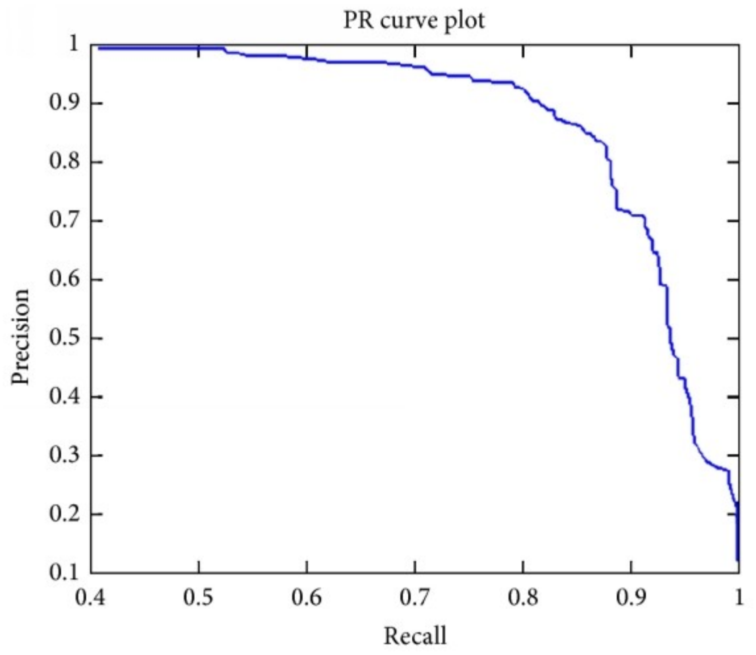
```
IMCONFS tensor([[24052.,    99.],
        [  391.,  1058.]])
Accuracy: tensor(0.9809)
Confusion tensor([[ 1058.,   391.],
        [   99., 24052.]])
Diced tensor(0.4278)Pixel Accuracy tensor(0.8630)
```

Average Precision (AP)

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

- AP is the area under the precision vs recall curve.



GPUs Error

- If the program can’t find your GPU uninstall pytorch and all its packages and install them again. Make sure to install the right CUDA version (This is what wasn’t able to use the GPU).