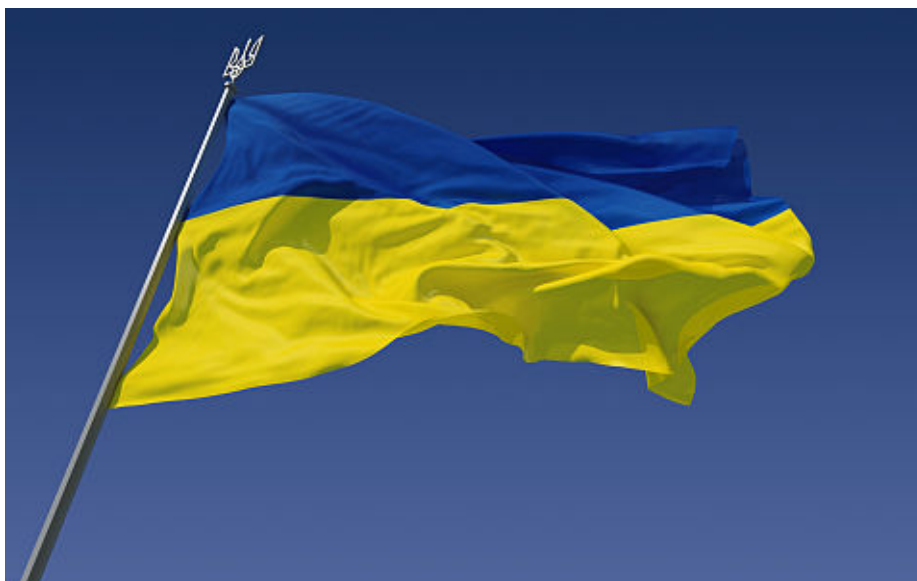

Important Update



On February 24th, 2022, Russia declared war and invaded peaceful Ukraine. After the annexation of Crimea and the occupation of the Donbas region, Putin's regime decided to destroy Ukrainian nationality. Ukrainians show fierce resistance and demonstrate to the entire world what it's like to fight for the nation's independence.

Ukraine's government launched a website to help russian mothers, wives & sisters find their beloved ones killed or captured in Ukraine - <https://200rf.com> & https://t.me/rf200_now (Telegram channel). Our goal is to inform those still in Russia & Belarus, so they refuse to assault Ukraine.

Help us get maximum exposure to what is happening in Ukraine, violence, and inhuman acts of terror that the "Russian World" has brought to Ukraine. This is a comprehensive Wiki on how you can help end this war: <https://how-to-help-ukraine-now.super.site/>

Official channels * Official account of the Parliament of Ukraine * Ministry of Defence * Office of the president * Cabinet of Ministers of Ukraine * Center of strategic communications * Minister of Foreign Affairs of Ukraine

Glory to Ukraine!

Pytorch-toolbelt

A `pytorch-toolbelt` is a Python library with a set of bells and whistles for PyTorch for fast R&D prototyping and Kaggle farming:

What's inside

- Easy model building using flexible encoder-decoder architecture.
- Modules: CoordConv, SCSE, Hypercolumn, Depthwise separable convolution and more.
- GPU-friendly test-time augmentation TTA for segmentation and classification
- GPU-friendly inference on huge (5000x5000) images
- Every-day common routines (fix/restore random seed, filesystem utils, metrics)
- Losses: BinaryFocalLoss, Focal, ReducedFocal, Lovasz, Jaccard and Dice losses, Wing Loss and more.
- Extras for Catalyst library (Visualization of batch predictions, additional metrics)

Showcase: Catalyst, Albumentations, Pytorch Toolbelt example: Semantic Segmentation @ CamVid

Why

Honest answer is “I needed a convenient way to re-use code for my Kaggle career”. During 2018 I achieved a Kaggle Master badge and this been a long path. Very often I found myself re-using most of the old pipelines over and over again. At some point it crystallized into this repository.

This lib is not meant to replace catalyst / ignite / fast.ai high-level frameworks. Instead it's designed to complement them.

Installation

```
pip install pytorch_toolbelt
```

How do I ...

Model creation

Create Encoder-Decoder U-Net model

Below a code snippet that creates vanilla U-Net model for binary segmentation. By design, both encoder and decoder produces a list of tensors, from fine (high-resolution, indexed 0) to coarse (low-resolution) feature maps. Access to all intermediate feature maps is beneficial if you want to apply

deep supervision losses on them or encoder-decoder of object detection task, where access to intermediate feature maps is necessary.

```
1 from torch import nn
2 from pytorch_toolbelt.modules import encoders as E
3 from pytorch_toolbelt.modules import decoders as D
4
5 class UNet(nn.Module):
6     def __init__(self, input_channels, num_classes):
7         super().__init__()
8         self.encoder = E.UNetEncoder(in_channels=input_channels,
9                                     out_channels=32, growth_factor=2)
9         self.decoder = D.UNetDecoder(self.encoder.channels,
10                                     decoder_features=32)
10        self.logits = nn.Conv2d(self.decoder.channels[0], num_classes,
11                                kernel_size=1)
11
12    def forward(self, x):
13        x = self.encoder(x)
14        x = self.decoder(x)
15        return self.logits(x[0])
```

Create Encoder-Decoder FPN model with pretrained encoder

Similarly to previous example, you can change decoder to FPN with contatenation.

```
1 from torch import nn
2 from pytorch_toolbelt.modules import encoders as E
3 from pytorch_toolbelt.modules import decoders as D
4
5 class SEResNeXt50FPN(nn.Module):
6     def __init__(self, num_classes, fpn_channels):
7         super().__init__()
8         self.encoder = E.SEResNeXt50Encoder()
9         self.decoder = D.FPNCatDecoder(self.encoder.channels,
10                                     fpn_channels)
10        self.logits = nn.Conv2d(self.decoder.channels[0], num_classes,
11                                kernel_size=1)
11
12    def forward(self, x):
13        x = self.encoder(x)
14        x = self.decoder(x)
15        return self.logits(x[0])
```

Change number of input channels for the Encoder

All encoders from `pytorch_toolbelt` supports changing number of input channels. Simply call `encoder.change_input_channels(num_channels)` and first convolution layer will be changed. Whenever possible, existing weights of convolutional layer will be re-used (in case new number of channels is greater than default, new weight tensor will be padded with randomly-initialized weights). Class method returns `self`, so this call can be chained.

```
1 from pytorch_toolbelt.modules import encoders as E
2
3 encoder = E.SEResnet101Encoder()
4 encoder = encoder.change_input_channels(6)
```

Misc

Count number of parameters in encoder/decoder and other modules

When designing a model and optimizing number of features in neural network, I found it's quite useful to print number of parameters in high-level blocks (like `encoder` and `decoder`). Here is how to do it with `pytorch_toolbelt`:

```
1 from torch import nn
2 from pytorch_toolbelt.modules import encoders as E
3 from pytorch_toolbelt.modules import decoders as D
4 from pytorch_toolbelt.utils import count_parameters
5
6 class SEResNeXt50FPN(nn.Module):
7     def __init__(self, num_classes, fpn_channels):
8         super().__init__()
9         self.encoder = E.SEResNeXt50Encoder()
10        self.decoder = D.FPNCatDecoder(self.encoder.channels,
11                                       fpn_channels)
12        self.logits = nn.Conv2d(self.decoder.channels[0], num_classes,
13                                 kernel_size=1)
14
15    def forward(self, x):
16        x = self.encoder(x)
17        x = self.decoder(x)
18        return self.logits(x[0])
19
20 net = SEResNeXt50FPN(1, 128)
21 print(count_parameters(net))
22 # Prints {'total': 34232561, 'trainable': 34232561, 'encoder':
23         25510896, 'decoder': 8721536, 'logits': 129}
```

Compose multiple losses

There are multiple ways to combine multiple losses, and high-level DL frameworks like Catalyst offers way more flexible way to achieve this, but here's 100%-pure PyTorch implementation of mine:

```
1 from pytorch_toolbelt import losses as L
2
3 # Creates a loss function that is a weighted sum of focal loss
4 # and lovasz loss with weights 1.0 and 0.5 accordingly.
5 loss = L.JointLoss(L.FocalLoss(), L.LovaszLoss(), 1.0, 0.5)
```

TTA / Inferencing

Apply Test-time augmentation (TTA) for the model

Test-time augmetnation (TTA) can be used in both training and testing phases.

```
1 from pytorch_toolbelt.inference import tta
2
3 model = UNet()
4
5 # Truly functional TTA for image classification using horizontal flips:
6 logits = tta.flip_lr_image2label(model, input)
7
8 # Truly functional TTA for image segmentation using D4 augmentation:
9 logits = tta.d4_image2mask(model, input)
```

Inference on huge images:

Quite often, there is a need to perform image segmentation for enormously big image (5000px and more). There are a few problems with such a big pixel arrays: 1. There are size limitations on maximum size of CUDA tensors (Concrete numbers depends on driver and GPU version) 2. Heavy CNNs architectures may eat up all available GPU memory with ease when inferencing relatively small 1024x1024 images, leaving no room to bigger image resolution.

One of the solutions is to slice input image into tiles (optionally overlapping) and feed each through model and concatenate the results back. In this way you can guarantee upper limit of GPU ram usage, while keeping ability to process arbitrary-sized images on GPU.

```
1 import numpy as np
2 from torch.utils.data import DataLoader
3 import cv2
4
```

```

5 from pytorch_toolbelt.inference.tiles import ImageSlicer,
   CudaTileMerger
6 from pytorch_toolbelt.utils.torch_utils import tensor_from_rgb_image,
   to_numpy
7
8
9 image = cv2.imread('really_huge_image.jpg')
10 model = get_model(...)
11
12 # Cut large image into overlapping tiles
13 tiler = ImageSlicer(image.shape, tile_size=(512, 512), tile_step=(256,
   256))
14
15 # HCW -> CHW. Optionally, do normalization here
16 tiles = [tensor_from_rgb_image(tile) for tile in tiler.split(image)]
17
18 # Allocate a CUDA buffer for holding entire mask
19 merger = CudaTileMerger(tiler.target_shape, 1, tiler.weight)
20
21 # Run predictions for tiles and accumulate them
22 for tiles_batch, coords_batch in DataLoader(list(zip(tiles, tiler.crops
   )), batch_size=8, pin_memory=True):
23     tiles_batch = tiles_batch.float().cuda()
24     pred_batch = model(tiles_batch)
25
26     merger.integrate_batch(pred_batch, coords_batch)
27
28 # Normalize accumulated mask and convert back to numpy
29 merged_mask = np.moveaxis(to_numpy(merger.merge()), 0, -1).astype(np.
   uint8)
30 merged_mask = tiler.crop_to_orignal_size(merged_mask)

```

Advanced examples

1. Inria Sattelite Segmentation
2. CamVid Semantic Segmentation

Citation

```

1 @misc{Khvedchenya_Eugene_2019_PyTorch_Toolbelt,
2   author = {Khvedchenya, Eugene},
3   title = {PyTorch Toolbelt},
4   year = {2019},
5   publisher = {GitHub},
6   journal = {GitHub repository},
7   howpublished = {\url{https://github.com/BloodAxe/pytorch-toolbelt}},
8   commit = {cc5e9973cdb0dcbf1c6b6e1401bf44b9c69e13f3}

```

