Introduction

I usually start each new solution by creating a solid baseline from scratch, without looking to provided baselines. To my beliefs, it helps to build a deep understanding of the problem. Although my experience in competitive machine learning is quite extensive, monocular geo-pose estimation from satellite imagery was new to me. After doing some initial EDA, it became clear that data presents a few challenges to participants: * Speckle noise in the AGL map (Presumably due to way of data acquisition) * Temporal inconsistency between measured AGL values and RGB images (A good example is a fence that is visible in AGL, but missing in RGB domain, different constructions sites) * Possible data leak due to overlapping/nearby tiles from same location belonging to train/test.

I've done my experiments on my self-built machines: One with 2080Ti and another PC with two 3090-s. I've used a single GPU PC for quick experiments and used the second for long-running training sessions in distributed mode. Training of a single model could easily take up to 3 days even on two 3090 cards.

Train/Validation split nuances

To my understanding, the safest way of splitting satellite imaging data into train/test parts is to ensure no city/region belongs to both train and test simultaneously. Checkerboard style of assigning image tiles to train and test is likely to cause severe data leak and make the validation untruthful.

In contrast, when using a location-based group split, we can guarantee that validation is leak-free and this gives us the possibility to evaluate model performance on the unseen regions.

Unfortunately, I've noticed that there is a certain degree of overlap / nearby tiles between train and test. So instead of building a location-based group split (Which is more correct from a business perspective), I've used the advantage of the data leak and did a location-stratified split instead. This way I intentionally used data leak since I knew beforehand it would be an advantage on the leaderboard compared to a leak-free split.

I've used a 10-fold CV split, but I've only managed to train 3 folds in the remaining amount of time. I expect a full 10-fold ensemble would perform substantially better.

Modelling approach

I’ve used a UNet-like model with two heads (AGL, MAG) attached to the output of the decoder (to predict AGL in meters and magnitude in centimeters) and a third head (ANGLE) attached to the output of the encoder to predict the orientation. The overall architecture mimics very much the official baseline model, but the structure of prediction heads differs significantly, which I will explain later. Unfortunately, I’ve started this competition just two weeks before the deadline and haven’t much time to do a proper ablation study. Comparing the performance of HRNet and FPN architectures to UNet would be a good idea for a future paper.

In my final solution I've used two model architectures: * EfficientNet B7 encoder with vanilla UNet decoder * EfficientNet B6 encoder with custom-made UNet-like decoder with residual deconvolutions.

Empirical study shows that diverse models play well in an ensemble. Therefore I intentionally was training different models on different folds to make the diverse ensemble. But two best-performing models were the heaviest ones from the above. Training each took almost 3 days on a 2x3090 setup.

Apart from optimized model architecture, I believe other key know-hows helped me to perform better in this challenge: - Good data augmentation to increase model robustness and generalization. It also allowed using test-time augmentation at the inference stage. - Tuning model architecture to have output stride 1 for predicted AGL - Proper choice of the loss function - Deep ensemble technique for blending predictions of separate models - Full D4 test-time augmentation

Height map predictions

Dense prediction heads (for AGL and Magnitude) were designed with gradual upsampling from stride 4 to stride 1. I found that simple bilinear upsampling to full resolution does not give enough accuracy. So I ended up bilinear upsampling operator by a factor of two, followed by convolution & activation. This block was repeated two times and 1x1 convolution at the end to make final predictions.

```python
class AGLHead(nn.Module):
    def __init__(
        self,
        encoder_channels: List[int],
        decoder_channels: List[int],
        embedding_size=64,
        dropout_rate=0.0,
        activation=ACT_RELU,
        num_upsample_blocks=1,
        agl_activation=ACT_RELU,
    ):
        super().__init__()
        in_channels = decoder_channels[0]
        self.dropout = nn.Dropout2d(dropout_rate, inplace=False)
        self.scale = ScaleHead()

        upsample_blocks = []
        for i in range(num_upsample_blocks):
            input_channels = (in_channels // 2) * 2
            upsampled_channels = input_channels // 4

            upsample_blocks += [
                conv1x1(in_channels, input_channels),
                ResidualDeconvolutionUpsample2d(input_channels, scale_factor=2),
                nn.Conv2d(upsampled_channels, upsampled_channels, kernel_size=3, padding=1, bias=False),
                nn.BatchNorm2d(upsampled_channels),
                instantiate_activation_block(activation, inplace=True),
            ]
            in_channels = upsampled_channels

        self.upsample = nn.Sequential(*upsample_blocks)

        self.height = nn.Sequential(
            nn.Conv2d(upsampled_channels + 1, upsampled_channels, kernel_size=3, padding=1),
            instantiate_activation_block(activation, inplace=True),
            nn.Conv2d(upsampled_channels, 1, kernel_size=3, padding=1),
            instantiate_activation_block(agl_activation, inplace=True),
        )

        self.magnitude = nn.Sequential(
            nn.Conv2d(upsampled_channels, upsampled_channels, kernel_size=3, padding=1),
            instantiate_activation_block(activation, inplace=True),
            nn.Conv2d(upsampled_channels, 1, kernel_size=3, padding=1),
            instantiate_activation_block(agl_activation, inplace=True),
        )

    def forward(self, rgb, encoder_feature_maps, decoder_feature_maps, gsd, orientation_outputs):
        # Take the finest feature map from the decoder
        x = decoder_feature_maps[0]
        x = self.upsample(x)

        gsd = gsd.reshape(gsd.size(0), 1, 1, 1).expand((-1, -1, x.size(2), x.size(3)))

        height = self.height(torch.cat([x, gsd], dim=1))
        mag = self.magnitude(x)
        scale = self.scale(mag, height)

        return {
            OUTPUT_AGL_MASK: height,
            OUTPUT_MAGNITUDE_MASK: mag,
            OUTPUT_VFLOW_SCALE: scale,
        }
```

AGL prediction head was also using the ground sample distance (GSD) value from the metadata file as an additional cue. Essentially it serves as a scaling coefficient to go from arbitrary units (inside CNN) to real units (meters).

MAG head did not have this input, since it predicts values in pixel units, and can deduct scaling on its own using only image data.

For optimizing this AGL & MAG objective I've tested many loss functions but ended up with Huber loss. According to the R2 score on validation, models trained with Huber loss performed better than the ones, trained with MSE or L1 or their combination. My explanation for this phenomenon is that Huber loss is less sensitive to outlier than MSE and converges faster than "vanilla" L1 loss. I tuned the gamma parameter to 10 for AGL and 5 to MAG to ensure there is still enough "sweet spot" for the MSE component of that loss.

It is also worth noting that I've used meter units as an objective for AGL optimization. The use of centimeter units leads to instabilities during the training in fp16 mode due to large values in target height maps. Scaling target values to meter units solved this problem. I've also experimented with log-scale height regression and log-cosh loss function, yet without noticeable improvement in R2 score.

Orientation prediction

For making predictions for orientation, I used similar approach as in the baseline solution - global pooling classifier attached to the last feature map from the encoder:

```
class BasicOrientationHead(nn.Module):
    def __init__(self, encoder_channels: List[int], decoder_channels: List[int], dropout_rate=0.0, activation=ACT_RELU):
        super().__init__()
        in_channels = encoder_channels[-1]
        self.pool = GlobalAvgPool2d(flatten=True)

        self.orientation = nn.Sequential(
            nn.Dropout(p=dropout_rate, inplace=True),
            nn.Linear(in_channels, in_channels),
            instantiate_activation_block(activation, inplace=True),
            nn.Linear(in_channels, 2),
        )

    def forward(self, rgb, encoder_feature_maps, decoder_feature_maps, gsd):
        features = self.pool(encoder_feature_maps[-1])
        direction = self.orientation(features)
        angle = tensor_vector2angle(direction)
        return {OUTPUT_VFLOW_DIRECTION: direction, OUTPUT_VFLOW_ANGLE: angle}
```

The only deviation from the baseline head - additional non-linearity for orientation regression to allow more dedication for angle representation of the head itself other than final layers of the encoder. Orientation was parametrized as vector of two components `cos(theta), sin(theta)`.

I've used a combination of Cosine and MSE losses for optimizing this objective. For the Cosine component, a weight of 1.0 was used, while MSE loss had the weight of 0.1 and acted as a regularization loss to keep predictions around the unit circle.

Also, it's worth noticing, that areas in the image, with small (<1m) ground height are unlikely to have enough signal in it to compute the orientation. Presumably if one can exclude or down-weight those areas during orientation prediction - this may increase the accuracy of the orientation prediction. One heuristics that came to my mind is the following: 1. Compute AGL predictions as usual 2. Compute the absolute magnitude of the spatial gradients of the predicted AGL. 3. Compute 2D softmax on abs. magnitude map to get an attention mask. 4. Compute orientation predictions in a dense fashion, similar to AGL head. 5. Multiply dense orientation predictions with an attention mask.

The line of thought is the following - this approach gives more weight to areas where height value is changing - edges of the buildings and disfavor flat regions.

Data Augmentation

Data augmentation proved itself to be a key ingredient in winning ML challenges. This one is not an exclusion. A well-known Albumentations library is now a de-facto standard library for image augmentation in python, which offers almost a hundred image/mask/boxes augmentation of all sorts. However, this competition contained data of many domains: - Spatial, pixels - Spatial, meters - Scalar, meters/pixel - Scalar, degree

It was necessary to have augmentations that were consistent across all domains. So I've implemented a custom affine augmentation class to augment heigh maps consistently with orientation angle and scale parameters. It helped to increase the diversity of image orientation and scale range during the training.

```
class GeoShiftScaleRotate(A.ShiftScaleRotate):
    mask_interpolation = cv2.INTER_LINEAR

    @property
    def targets(self):
        return {
            "image": self.apply,
            "mask": self.apply_to_mask,
            "masks": self.apply_to_masks,
            "bboxes": self.apply_to_bboxes,
            "keypoints": self.apply_to_keypoints,
            "gsd": self.apply_to_gsd,
            "vflow_scale": self.apply_to_vflow_scale,
            "vflow_angle": self.apply_to_vflow_angle,
        }

    def apply_to_mask(self, img, angle=0, scale=0, dx=0, dy=0, **params):
        return A.shift_scale_rotate(img, angle, scale, dx, dy, self.mask_interpolation, self.border_mode, self.mask_value)

    def apply_to_gsd(self, gsd, scale=0, **params):
        """
        Ground sample distance (GSD) in meters per pixel.
        GSD is the average pixel size in meters. Units - meters/pixel
        """

        # When the scale is < 1, crop patch is smaller than destination patch, so the number of meters in pixel decreases
        # When the scale is > 1, crop patch is larger than destination patch, so the number of meters in pixel increases
        return gsd * scale

    def apply_to_vflow_scale(self, vflow_scale, scale=0, **params):
        return vflow_scale / scale

    def apply_to_vflow_angle(self, vflow_angle, angle=0, **params):
        angle_rads = np.radians(angle)
        vflow_angle = vflow_angle + angle_rads

        if vflow_angle > math.pi * 2:
            vflow_angle -= math.pi * 2
        if vflow_angle < 0:
            vflow_angle += math.pi * 2
        return vflow_angle
```

I've also implemented augmentations for 90-degree rotation and transposition. Training with these augmentations increased model stability and enabled me to use D4 test-time augmentation at the inference stage.

Apart from spatial augmentations, I've also used photometric augmentations. In particular:

- Random brightness & contrast change
- Random tone curve adjustment
- Random gamma correction
- Applying different noise models: Gaussian, ISO, Multiplicicative
- Random RGB channels shift
- Random shifts in HSV colorspace
- Fancy PCA Augmentation
- Random fog augmentation
- Coarse dropout of square blocks up to 128x128

The full augmentation pipeline looked as follows:

```
def get_medium_augmentations(train_image_size: Tuple[int, int], ignore_label: int) -> List[A.DualTransform]:
    return [
        GeoRandomRotate90(p=1),
        GeoTranspose(p=0.5),
        GeoShiftScaleRotate(scale_limit=0.1, rotate_limit=22, value=0, mask_value=ignore_label, border_mode=cv2.BORDER_CONSTANT, p=0.
        A.OneOf([
            A.RandomBrightnessContrast(brightness_limit=0.1, contrast_limit=0.1),
            A.CLAHE(),
            A.RandomToneCurve(),
            A.RandomGamma(),
        ], p=0.5),
        # More color changes
        A.OneOf(
            [A.ISONoise(), A.GaussNoise(), A.MultiplicativeNoise(), A.RandomFog(fog_coef_lower=0.1, fog_coef_upper=0.5)], p=0.1
        ),
        A.OneOf([
            A.RGBShift(r_shift_limit=10, g_shift_limit=10, b_shift_limit=10),
            A.HueSaturationValue(hue_shift_limit=5, sat_shift_limit=5, val_shift_limit=5),
            A.FancyPCA(),
        ], p=0.1),
        A.CoarseDropout(
            max_holes=2,
            min_height=8,
            max_height=128,
            min_width=8,
            max_width=128,
            mask_fill_value=ignore_label,
            fill_value=0,
            p=0.1,
        ),
    ]
```

This set of augmentation isn't scientifically picked, but rather battle-tested in many similar competitions (Inria Aerial Labeling, Kaggle, SpaceNet challenges). Usually, as a rule of thumb - the heavier the model, the more augmentations you may apply to prevent overfitting.

According to training logs, I was very far from overfitting. But also, it could be a leaky validation scheme that I have deliberately chosen. Anyway, photometric augmentation usually helps a lot. Spatial augmentation should be applied with care, but also generally safe.

Inference tricks

During the training session, I've saved the best 3 checkpoints based on the R2 score on validation. At inference, I used all of them and averaged their predictions to get the final result. Averaging height map predictions and scale was done naively using the mean average. Angle averaging was done a bit differently - I normalized angle predictions to unit length and then averaged them.

The second inference trick was test-time augmentation. When properly trained, the model is invariant to flips/D2/D4 augmentations. However, averaging predictions of 8 variants of the input image, greatly reduces the variance of the predictions and improves a final R2 at a price of increased inference time. According to my measurements, D4 TTA gave a boost of ~0.005~0.01 R2 depending on the model.

All ensembling was done entirely on the GPU, without involving temporary storage. For this purpose I've written a special wrapper to make augmentation/deaugmentation on the fly during the inference:

```
 def geopose_model_d4_tta(model, agl_reduction="mean"):
     return GeoposeTTAModel(
         model,
         augment_fn={INPUT_IMAGE_KEY: d4_image_augment, INPUT_GROUND_SAMPLE_DISTANCE: d4_labels_augment},
         deaugment_fn={
             OUTPUT_VFLOW_SCALE: d4_labels_deaugment,
             OUTPUT_VFLOW_DIRECTION: d4_direction_deaugment,
             OUTPUT_AGL_MASK: partial(d4_image_deaugment, reduction=agl_reduction),
         },
     )

class GeoposeTTAModel(GeneralizedTTA):
    def forward(self, **kwargs):
        augmented_inputs = dict((key, augment(kwargs[key])) for (key, augment) in self.augment_fn.items())
        num_inputs = augmented_inputs[INPUT_IMAGE_KEY].size(0)
        outputs = []
        for i in range(num_inputs):
            outputs_i = self.model(**dict((key, value[i : i + 1]) for (key, value) in augmented_inputs.items()))
            outputs.append(outputs_i)

        deaugmented_outputs = {}
        for key, deaugment_fn in self.deaugment_fn.items():
            key_outputs = torch.cat([outputs_i[key] for outputs_i in outputs], dim=0)
            # Ensure that angle orientation is always normalized
            if key == OUTPUT_VFLOW_DIRECTION:
                key_outputs = F.normalize(key_outputs, dim=1)  # [N,B,C]

            output = deaugment_fn(key_outputs)
            deaugmented_outputs[key] = output

        deaugmented_outputs[OUTPUT_VFLOW_ANGLE] = tensor_vector2angle(deaugmented_outputs[OUTPUT_VFLOW_DIRECTION])
        return deaugmented_outputs
```

I've found TTA to be an extremely useful way of "cheap" increase of model accuracy. Yet it comes for N times increase in inference time (8 for D4, 4 for D2, and 2x for horizontal flips). Yet in uncapped competitions, participants are very tempted to go with huge ensembles and heavy TTA techniques as long as rules do not restrict inference time.

What didn't work or I didn't have time to play with

Next time I will start a bit earlier. Two weeks was indeed a “last call” to hop on. The challenge was complicated and hardware demanding. Also, there were many strong participants on the leaderboard. A few ideas left intact this time: - Transformers-based model. It has been shown recently, that ViT, SWIN, and other approaches can beat classical CNN models in segmentation tasks. Therefore, it would be interesting to compare their performance with my solution. - Trimmed losses. Monocular depth estimation benefits greatly from using trimmed losses to exclude the influence of strong outliers during the training. I wonder whether this can be applied to geopose estimation as well. - HR-Net, FPN architectures, NFNets - GAN-based data augmentation. I feel this is a largely underestimated technique. It requires significant effort to master but can lead to a substantial boost of the model generalization. - Direct optimization of R2 score. Could be interesting to look at how better/worse it will be compared to MSE/Huber losses.

I'd like to say thanks to the organizers of this challenge for such an interesting task! Hope to see more similar competitions in the future.

References

- U-Net: Convolutional Networks for Biomedical Image Segmentation (https://arxiv.org/abs/1505.04597)
- Albumentations (https://github.com/albumentations-team/albumentations)
- Catalyst (https://github.com/BloodAxe/xView2-Solution)
- pytorch-toolbelt (https://github.com/BloodAxe/pytorch-toolbelt)
- XView2 Solution (https://github.com/BloodAxe/xView2-Solution)
- GeoPose Solution (https://github.com/BloodAxe/DrivenData-2021-Geopose-Solution)
- PyTorch Image Models (https://github.com/rwightman/pytorch-image-models)
- The Devil is in the Decoder: Classification, Regression and GANs (https://arxiv.org/abs/1707.05847)
- Learning Geocentric Object Pose in Oblique Monocular Images (https://openaccess.thecvf.com/content_CVPR_2020/papers/Christie_Learning_Geocentric_Object_Pose_in_Oblique_Monocular_Images_CVPR_2020_paper.pdf)