Paper review

Paper info

Title: HoloGAN: Unsupervised learning of 3D representations from natural images
Authors: Thu Nguyen-Phuoc, Chuan Li, Lucas Theis, Christian Richardt, Yong-Liang Yang
Link: https://arxiv.org/abs/1904.01326
Tags: Computer Vision, GAN, unsupervised learning
Year: 2019
Code:

Summary

What

- conditional GANs for understanding 3D strucutres requires a lot of labelled data for training, however 3D ground-truth data are very expensive to capture and reconstruct;
- there is suggested an architecture (HoloGAN) that allows unsupervised learning of 3D representations directly from natural images;
- after learning 3D structure and understanding a target pose, HoloGAN is able to generate new views of the same scene.

How

1) Generator
Main difference with typical GANs: HoloGAN tries to understand 3D representation of the world and then apply 3D rigid-body transformation for the found representation while typical GANs learn to map a noise input vector directly to 2D features to generate images. As a result, in HoloGAN a strong inductive bias about the 3D world is added into the generator network.

- on the first stage HoloGAN tries to understand 3D representation, e.g. disentangle pose and identity. HoloGAN learns 3D features from a 4D constant tensor (size 4x4x4x512);
- HoloGAN performs explicit 3D rigid-body transformation (3D rotation followed by trilinear resampling);
- projection unit generates 2D images (128x128) from 3D structure.

2) Discriminator: in addition to the image discriminator that classifies images as real or fake there is proposed a multi-scale style discriminators that classifies the same at the feature level.

Datasets details: all datasets except one contain unique single views.
Training details: Adam solver used for trainining process.

Results:
1) demonstrated that generate views with different azimuth keeping the same identities for different images;
2) Kernel Inception Distance (KID) was used to evaluate the visual fidelity. HoloGAN demonstrates competitive or better results than other recent GAN models: DCGAN, LSGAN and WGAN-GP.
3) On the one dataset, it is shown that HoloGAN can generate images better than state-of-the-art visual object networks (VON), in particular, HoloGAN can produce images in full 360° views, while VON struggles to create images from the back views.
4) It is visually shown that HoloGAN can disentangle identity and pose;
5) It is visually shown that randomly rotating the 3D features during training is crucial for HoloGAN; otherwise, it will fail to generate images with different poses.

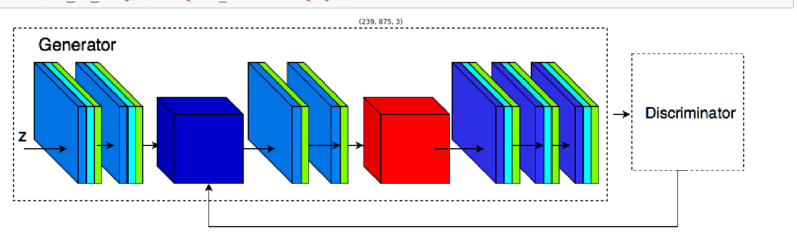
Conclusion: the main advantage of HoloGan is that it doesn't require expensive labeled data and demonstrate competitive or even better results in comparison with state-of-the-art architectures.

CNN visualization

HoloGAN architecture is presented on the following plot:

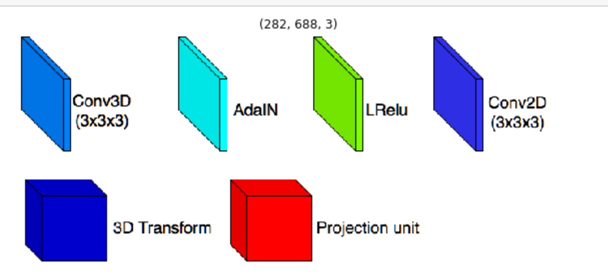
In [8]:
```
imshow(url_to_image("./report_files/cnn.png"))
```

(239, 875, 3)



In [14]:
```
imshow(url_to_image("./report_files/vocabular.png"))
```

(282, 688, 3)



Experiment summary

Experiments with baseline model

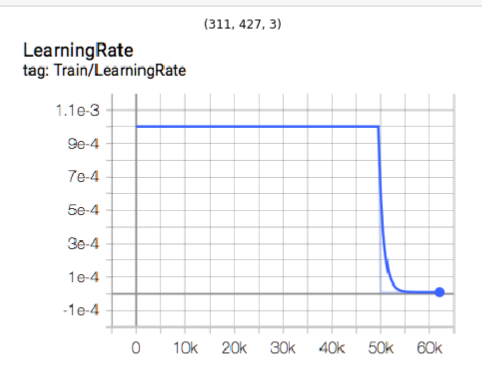
Train baseline model

I run initial baseline model as is and received the following results:

This plot shows how learning rate changes per iteration.

In [32]:
```
imshow(url_to_image("./report_files/baseline_lr_iteration.png"))
```

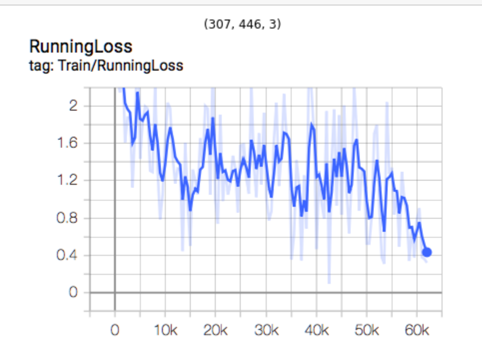
(311, 427, 3)



This plot shows how training loss changes per iteration.

In [33]:
```
imshow(url_to_image("./report_files/baseline_loss_iteration.png"))
```

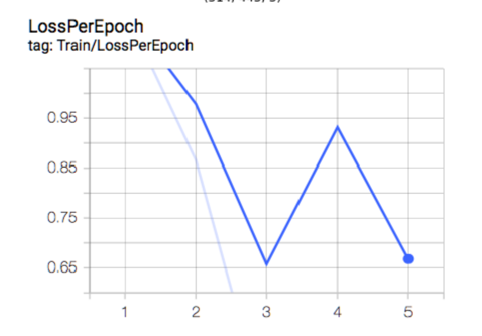
(307, 446, 3)



This plot shows how training loss changes per epoch.

In [34]:
```
imshow(url_to_image("./report_files/baseline_loss_epoch.png"))
```

(314, 443, 3)



Test accuracy of baseline model is **64%**.

Baseline model with augmentation

I used 'imgaug' (https://github.com/aleju/imgaug) library in order to add custom augmentation. I run a couple of experiments with baseline model + augmentation.
"Small" augmentation (just crop and affine transformation):

In [35]:
```
class ImgAugTransform:
  def __init__(self):
    # Sometimes(0.5, ...) applies the given augmenter in 50% of all cases,
    # e.g. Sometimes(0.5, GaussianBlur(0.3)) would blur roughly every second image.
    sometimes = lambda aug: iaa.Sometimes(0.5, aug)

    # Define our sequence of augmentation steps that will be applied to every image
    # All augmenters with per_channel=0.5 will sample one value _per image_
    # in 50% of all cases. In all other cases they will sample new values
    # _per channel_.
    self.aug = iaa.Sequential(
        [
            # apply the following augmenters to most images
            # crop images by -5% to 10% of their height/width
            sometimes(iaa.CropAndPad(
                percent=(-0.05, 0.1),
                pad_mode=ia.ALL,
                pad_cval=(0, 255)
            )),
            sometimes(iaa.Affine(
                scale={"x": (0.8, 1.2), "y": (0.8, 1.2)}, # scale images to 80-120% of their size, i
ndividually per axis
                translate_percent={"x": (-0.2, 0.2), "y": (-0.2, 0.2)}, # translate by -20 to +20 pe
rcent (per axis)
                rotate=(-45, 45), # rotate by -45 to +45 degrees
                shear=(-16, 16), # shear by -16 to +16 degrees
                order=[0, 1], # use nearest neighbour or bilinear interpolation (fast)
                cval=(0, 255), # if mode is constant, use a cval between 0 and 255
                mode=ia.ALL # use any of scikit-image's warping modes (see 2nd image from the top fo
r examples)
            )),
        ],
        random_order=True
    )

  def __call__(self, img):
    img = np.array(img)
    return self.aug.augment_image(img)
```

Test accuracy of baseline model + small augmentation is **64% - not improved**.

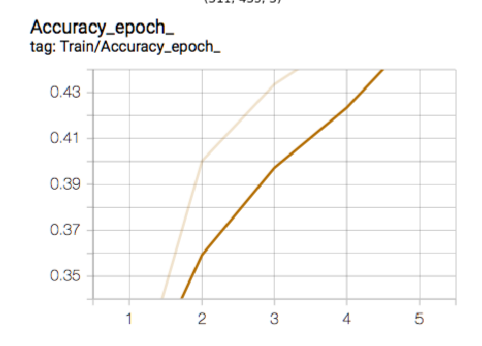
I also tried to use more significant (blurring, sharpen, emboss, gaussian noise, etc - more details in jupyter notebook) augmentation and as a result got:
test accuracy of baseline model + significant augmentation is **50%**.
The possible reasons of accuracy decrease are:

- Augmentation changes images too significantly;
- The baseline model is pretty simple, so no overfitting here and augmentation isn't needed.

This plot shows how training accuracy changes per epoch.

In [37]:
```
imshow(url_to_image("./report_files/baseline_aug_train_acc_epoch.png"))
```

(311, 433, 3)



Baseline model with different learning rate

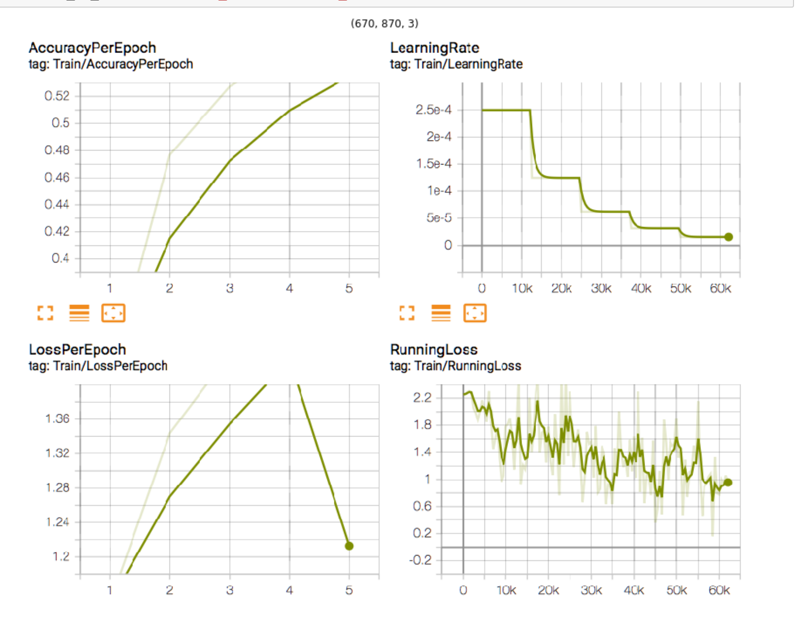
I changed learning rate according to the following algorithm: decrease learning rate on 50% on every epoch.

In [ ]:
```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                            step_size=1,
                                            gamma=0.5)
```

This plot shows results of baseline model + learning rate schedule (without augmentation).

In [42]:
```
imshow(url_to_image("./report_files/baseline_lr.png"))
```

(670, 870, 3)



Test accuracy in this case is **60%** - not improved in comparison with baseline model.

Experiments with improved model¶

Improved model architecture

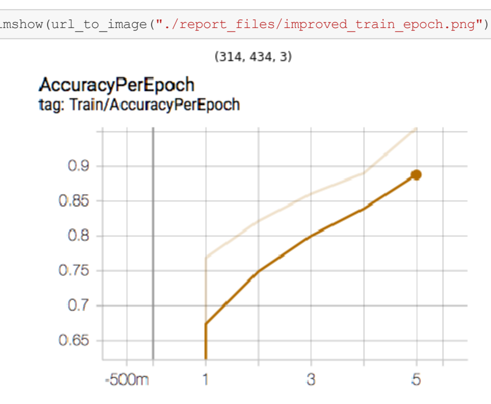
Model architecture has the following structure (based on https://zhenye-na.github.io/2018/09/28/pytorch-cnn-cifar10.html):

In [ ]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv_layer = nn.Sequential(

            # Conv Layer block 1
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Conv Layer block 2
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout2d(p=0.05),

            # Conv Layer block 3
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.fc_layer = nn.Sequential(
            nn.Dropout(p=0.1),
            nn.Linear(4096, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.1),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        # conv layers
        x = self.conv_layer(x)
        # flatten
        x = x.view(x.size(0), -1)
        # fc layer
        x = self.fc_layer(x)
        return x
```

Train accuracy (per epoch) is shown on the following plot:

In [66]:
```
imshow(url_to_image("./report_files/improved_train_epoch.png"))
```

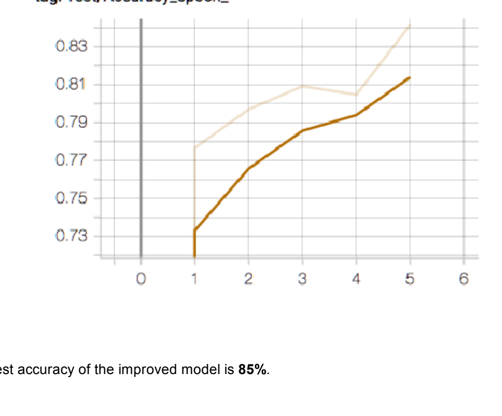
(314, 434, 3)



Test accuracy (per epoch) is shown on the following plot:

In [55]:
```
imshow(url_to_image("./report_files/improved_test_epoch.png"))
```

(303, 431, 3)



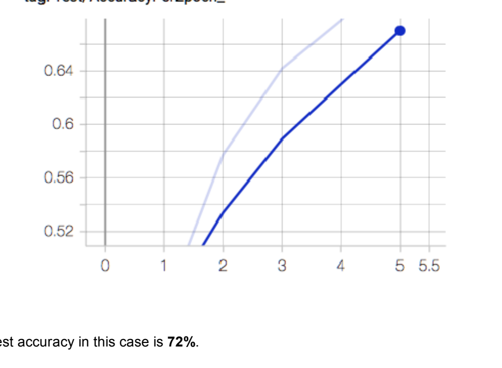
Test accuracy of the improved model is **85%**.

Improved model architecture + augmentation + different learning rate

I also run a series of experiments with improved model + augmentation + different learning (similar to how I did it in baseline model). But in these experiments I didn't get better results.
For example, test accuracy (per epoch) of the improved model with augmentation is shown on the following plot:

In [58]:
```
imshow(url_to_image("./report_files/improved_aug_test_acc.png"))
```

(319, 435, 3)



Test accuracy in this case is **72%**.

Compare time on CPU and GPU

Time for 5 epochs training:
- ~1h 5min on CPU;
- ~6min 36second on GPU (Tesla P4);

Conclusions

The best achieved accuracy on the test dataset is 85%. Of course, this result can be improved.
The most perspective possible steps for improvement:

- Use other architecture (googlenet, mobilenet, VGG, ResNet, etc);
- Use transfer learning;
- More experiments with hyperparameters, use different optimizers;
- Train for more epochs, it seems that 5 epochs isn't enough for the final architecture.