Homework 0 Wenbo Hu A15870455

Problem 1

1. Gradient of Lagrangian

$$\frac{1}{2}\frac{d}{dx}((\mathbf{Ax}-\mathbf{b})^\top(Ax-b))+2\lambda x$$
$$=\frac{d}{dx}(\mathbf{x}^\top\mathbf{A}^\top Ax-2(\mathbf{b}^\top A)x+\mathbf{b}^\top b)+2\lambda x$$
$$=\mathbf{A}^\top Ax-\mathbf{A}^\top b+2\lambda x$$

2. Unconstrained least square

$$x=(A^TA)^\dagger A^Tb$$

3.a

$$\mathbf{A}^\top Ax-\mathbf{A}^\top b+2\lambda x=0$$
$$\mathbf{A}^\top b=\mathbf{A}^\top Ax+2\lambda x$$
$$x=(A^TA+2\lambda I)^\dagger A^Tb$$

3.b

$$h(\lambda)^Th(\lambda)=b^TA[(A^TA+2\lambda I)^\dagger]^T(A^TA+2\lambda I)^\dagger A^Tb$$
$$let, A^TA=UDU^T$$
$$so,(A^TA+2\lambda I)^\dagger=U(D+2\lambda I)^{-1}U^T=UBU^T$$
$$then, h(\lambda)^Th(\lambda)=b^TA(UBU^T)^TUBU^TA^Tb$$
$$=b^TAUBBU^TA^Tb$$
$$=b^TAUBB(b^TAU)^T$$

so this is a positive semidefinite matrix for $\lambda \geq 0$, it's monotonically decreasing

4. Implement

In [143]:
```
import numpy as np
npz = np.load('HW0_P1.npz')
A = npz['A']
b = npz['b']
eps = npz['eps']
A.shape, A.dtype, b.shape, b.dtype
```

Out[143]:
```
((100, 30), dtype('float64'), (100,), dtype('float64'))
```

In [139]:

In [152]:
```
def solve(A, b, eps):
    # your implementation here
    start = 1e-2
    while True:
        x = np.linalg.pinv(A.T@A + 2*start*np.eye(30)) @A.T @ b
        f_x = x.T @ x
        if np.abs(f_x - eps) < 1e-6:
            return x
        elif f_x < eps:
            start = start / 2
        else:
            start = start * (3/2)
    return x
```

In [153]:
```
# Evaluation code, you need to run it, but do not modify
x = solve(A,b,eps)
print('x norm square', x@x)  # x@x should be close to or less then eps
print('optimal value', ((A@x - b)**2).sum())
```

```
x norm square 0.4999994990344474
optimal value 17.22012797060903
```

Problem 2

(2.1)

$$Pr(P \leqslant t) = \iint p(\alpha^{'} A + \beta^{'} B \leqslant t) d\alpha d\beta$$
$$= \iint p(\frac{\beta}{\alpha + \beta} \leqslant t) d\alpha d\beta$$
$$since\,, \beta \leqslant \alpha t + \beta t$$
$$= \int d\beta \int p(\beta \leqslant \frac{t}{1-t}\alpha) d\alpha$$
$$when, \frac{t}{1-t} \leqslant 1 \quad t \in [0, 0.5]$$
$$F(t) = \int p(\beta \leqslant \frac{t}{1-t}\alpha) d\alpha = \frac{1}{2}\frac{t}{1-t}$$
$$when, t\ > 1, \quad t \in (0.5, 1]$$
$$F(t) = \int p(\beta \leqslant \frac{t}{1-t}\alpha) d\alpha = \frac{3t-1}{2t}$$
$$\int_0^{\frac{1-t}{t}} \frac{t}{1-t}\alpha d\alpha + \int_{\frac{1-t}{t}}^{1} 1 d\alpha$$
$$= \frac{1}{2} * \frac{1-t}{t} + (1 - \frac{1-t}{t})$$
$$= \frac{1 - t + 2(2t-1)}{2t} = \frac{3t-1}{2t}$$

(2.1)

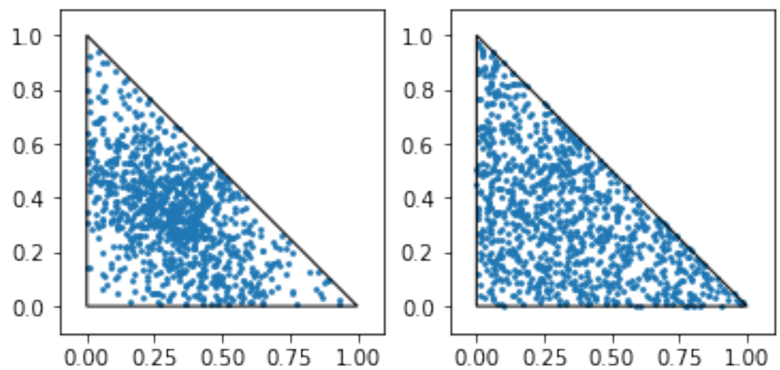
$$F^{'}(t) = \begin{cases} \frac{1}{2}[t(1-t)^{-2} + (1-t)^(- 1)] = \frac{1}{2}\frac{t+1-t}{(1-t)^2} = \frac{1}{2(1-t)^2} & t \in [0, 0.5] \\ 3\frac{1}{2t} + \frac{3t-1}{2}(-t^{-2}) = \frac{3t-3t+1}{2t^2} = \frac{1}{2t} & t \in (0.5, 1] \end{cases}$$
$$\mathrm{P} = 0 \quad \frac{1}{2}$$
$$\mathrm{P} = 0 \quad 2$$

(2.2)

$$Y = A + \alpha(B - A) + \beta(C - A),$$
$$r.v. \quad x = \left[\alpha, \beta\right]$$
$$let\ T = \left| \overrightarrow{AB}\ \overrightarrow{AC} \right| \ \ \overrightarrow{OA} = \vec{b}$$
$$Y = \vec{b} + Tx$$
$$since\ \ g(y) = f(H^{-1}(y))\left|det(J^{-1})\right|$$
$$Pr(y) = P_x(T^{-1}(y - \vec{b})) \cdot \left|det(T^{-1})\right|$$
$$since Pr(\alpha, \beta) = Pr(\alpha) \cdot Pr(\beta)$$
$$P(x) = \frac{1}{1-0} \cdot \frac{1}{1-0} = 1$$
$$Pr(y) = (Tx + b)^{-1}(Tx + b)\left|det(T^{-1})\right|$$
$$= \left|det(T^{-1})\right|$$
$$\text{which is in the genel pdf form} \frac{1}{b-a}$$
$$\text{so it's uniform distribution}$$

In [115]:
```
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon

pts = np.array([[0,0], [0,1], [1,0]])
def draw_background(index):
    # DRAW THE TRIANGLE AS BACKGROUND
    p = Polygon(pts, closed=True, facecolor=(1,1,1,0), edgecolor=(0, 0

    plt.subplot(1, 2, index + 1)

    ax = plt.gca()
    ax.set_aspect('equal')
    ax.add_patch(p)
    ax.set_xlim(-0.1,1.1)
    ax.set_ylim(-0.1,1.1)

A = np.array([0,0])
B = np.array([0,1])
C = np.array([1,0])
wrong = []
for i in range(1000):
    alpha, beta, gamma = np.random.uniform(0,1,3)
    p = alpha / (alpha+beta+gamma) * A + beta / (alpha+beta+gamma) * B
    wrong.append(p)
right = []
for i in range(1000):
    alpha, beta = np.random.uniform(0,1,2)
```

```
    alpha, beta = np.random.uniform(0,1,2)
    p = A + alpha *(B-A) + beta * (C-A)
    if -p[0] + 1 < p[1]:
        p = B + C - p
    right.append(p)

draw_background(0)
# REPLACE THE FOLLOWING LINE USING YOUR DATA (incorrect method)
plt.scatter(np.array(wrong)[:,0], np.array(wrong)[:,1], s=3)

draw_background(1)
# REPLACE THE FOLLOWING LINE USING YOUR DATA (correct method)
plt.scatter(np.array(right)[:,0], np.array(right)[:,1], s=3)

plt.show()
```

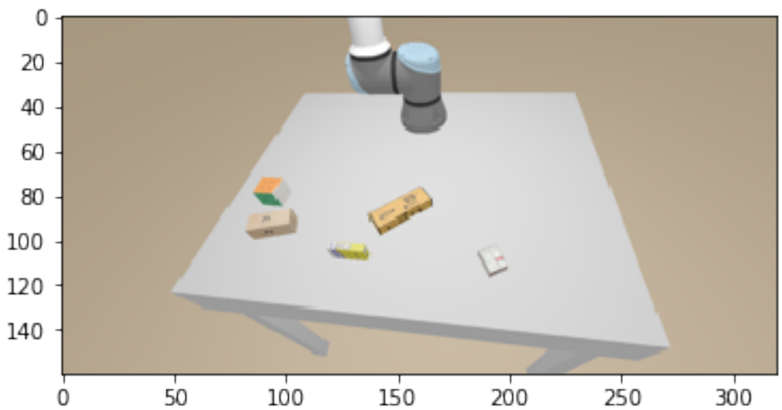


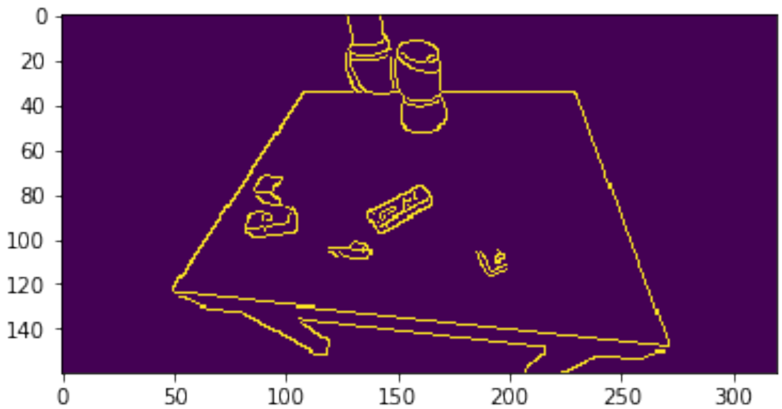
Problem 3

In [71]:
```
import numpy as np
npz = np.load("train.npz")
images = npz["images"]  # array with shape (N,Width,Height,3)
edges = npz["edges"]  # array with shape (N,Width,Height)
```

In [72]:
```
plt.figure()
plt.imshow(images[0])
plt.figure()
plt.imshow(edges[0])
```

Out[72]: <matplotlib.image.AxesImage at 0x7febd7b8d190>





In [73]:
```
images.shape, edges.shape, images.max(), np.unique(edges)
```

Out[73]: ((1000, 160, 320, 3), (1000, 160, 320), 255, array([0, 255], dtype=uint8))

In [74]:
```
edges = np.expand_dims(edges, axis=0)
images = images.transpose((0, 3, 1, 2))
edges = edges.transpose((1, 0, 2, 3))
```

In [76]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F


class DoubleConv(nn.Module):

    def __init__(self, in_channels, out_channels, mid_channels=None):
```

```
    def __init__(self, in_channels, out_channels, mid_channels=None):
        super().__init__()
        if not mid_channels:
            mid_channels = out_channels
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, kernel_size=3, paddin
            nn.BatchNorm2d(mid_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(mid_channels, out_channels, kernel_size=3, paddi
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        return self.double_conv(x)


class Down(nn.Module):

    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )

    def forward(self, x):
        return self.maxpool_conv(x)


class Up(nn.Module):

    def __init__(self, in_channels, out_channels, bilinear=True):
        super().__init__()

        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear', ali
            self.conv = DoubleConv(in_channels, out_channels, in_chann
        else:
            self.up = nn.ConvTranspose2d(in_channels, in_channels // 2
            self.conv = DoubleConv(in_channels, out_channels)

    def forward(self, x1, x2):
        x1 = self.up(x1)
        # input is CHW
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2,
                        diffY // 2, diffY - diffY // 2])
        x = torch.cat([x2, x1], dim=1)
```

```
        return self.conv(x)


class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1

    def forward(self, x):
        return self.conv(x)

class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, bilinear=True):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        factor = 2 if bilinear else 1
        self.down4 = Down(512, 1024 // factor)
        self.up1 = Up(1024, 512 // factor, bilinear)
        self.up2 = Up(512, 256 // factor, bilinear)
        self.up3 = Up(256, 128 // factor, bilinear)
        self.up4 = Up(128, 64, bilinear)
        self.outc = OutConv(64, n_classes)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits
```

In [77]:
```
batch_size = 20
import torch.utils.data as utils
data_loaders = []
images_train = images[:800]
edges_train = edges[:800]
images_valid = images[800:]
edges_valid = edges[800:]

for (data, edge) in [(images_train, edges_train), (images_valid, edges

    imgs = torch.tensor(data).float().contiguous()
    imgs = imgs / 255

    edge = torch.tensor(edge).long().contiguous()
    edge = edge / 255
    dataset = utils.TensorDataset(imgs,edge)
    dataloader = utils.DataLoader(dataset, batch_size=batch_size, shuf
    data_loaders.append(dataloader)

train_loader = data_loaders[0]
valid_loader = data_loaders[1]

dataloaders = {
    'train': train_loader,
    'val': valid_loader
}
```

In [78]:
```
# Get a batch of training data
inputs, masks = next(iter(train_loader))

print(inputs.shape, masks.shape)
for x in [inputs.numpy(), masks.numpy()]:
    print(x.min(), x.max(), x.mean(), x.std())
```

```
torch.Size([20, 3, 160, 320]) torch.Size([20, 1, 160, 320])
0.0 1.0 0.6927372 0.12354136
0.0 1.0 0.04060547 0.19737439
```

In [82]:
```
from torchsummary import summary
import torch
import torch.nn as nn

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model= UNet(n_channels=3, n_classes=1, bilinear=True)
model = model.to(device)

summary(model, input_size=(3, 160, 320))
```

In [80]:
```
from collections import defaultdict

def calc_loss(pred, target, metrics, bce_weight=1.0):
    bce = F.binary_cross_entropy_with_logits(pred, target)
    pred = F.sigmoid(pred)
    #dice = dice_loss(pred, target)
    dice = 0
    loss = bce * bce_weight + dice * (1 - bce_weight)

    metrics['bce'] += bce.data.cpu().numpy() * target.size(0)
    #metrics['dice'] += dice.data.cpu().numpy() * target.size(0)
    metrics['loss'] += loss.data.cpu().numpy() * target.size(0)

    return loss

def print_metrics(metrics, epoch_samples, phase):
    outputs = []
    for k in metrics.keys():
        outputs.append("{}: {:4f}".format(k, metrics[k] / epoch_sample

    print("{}: {}".format(phase, ", ".join(outputs)))

def train_model(model, optimizer, scheduler, num_epochs=5):
    best_model_wts = copy.deepcopy(model.state_dict())
    best_loss = 1e10

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

        since = time.time()

        for phase in ['train', 'val']:
            if phase == 'train':
                scheduler.step()
                for param_group in optimizer.param_groups:
                    print("LR", param_group['lr'])

                model.train()
            else:
                model.eval()

            metrics = defaultdict(float)
            epoch_samples = 0

            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                optimizer.zero_grad()
```

```
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    loss = calc_loss(outputs, labels, metrics)

                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                epoch_samples += inputs.size(0)

            print_metrics(metrics, epoch_samples, phase)
            epoch_loss = metrics['loss'] / epoch_samples

            if phase == 'val' and epoch_loss < best_loss:
                print("saving best model")
                best_loss = epoch_loss
                best_model_wts = copy.deepcopy(model.state_dict())

        time_elapsed = time.time() - since
        print('{:.0f}m {:.0f}s'.format(time_elapsed // 60, time_elapse
    print('Best val loss: {:4f}'.format(best_loss))

    model.load_state_dict(best_model_wts)
    return model
```

In [ ]:

```
import torch.optim as optim
from torch.optim import lr_scheduler
import time
import copy

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu"
print(device)

num_class = 1

model =  UNet(n_channels=3, n_classes=1, bilinear=True).to(device)

optimizer_ft = optim.Adam(model.parameters(), lr=1e-3)

exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=25, gam

model = train_model(model, optimizer_ft, exp_lr_scheduler, num_epochs=
```

In [86]:

```
model.load_state_dict(torch.load('hw0_model_weights_newunet.pth', map_
```

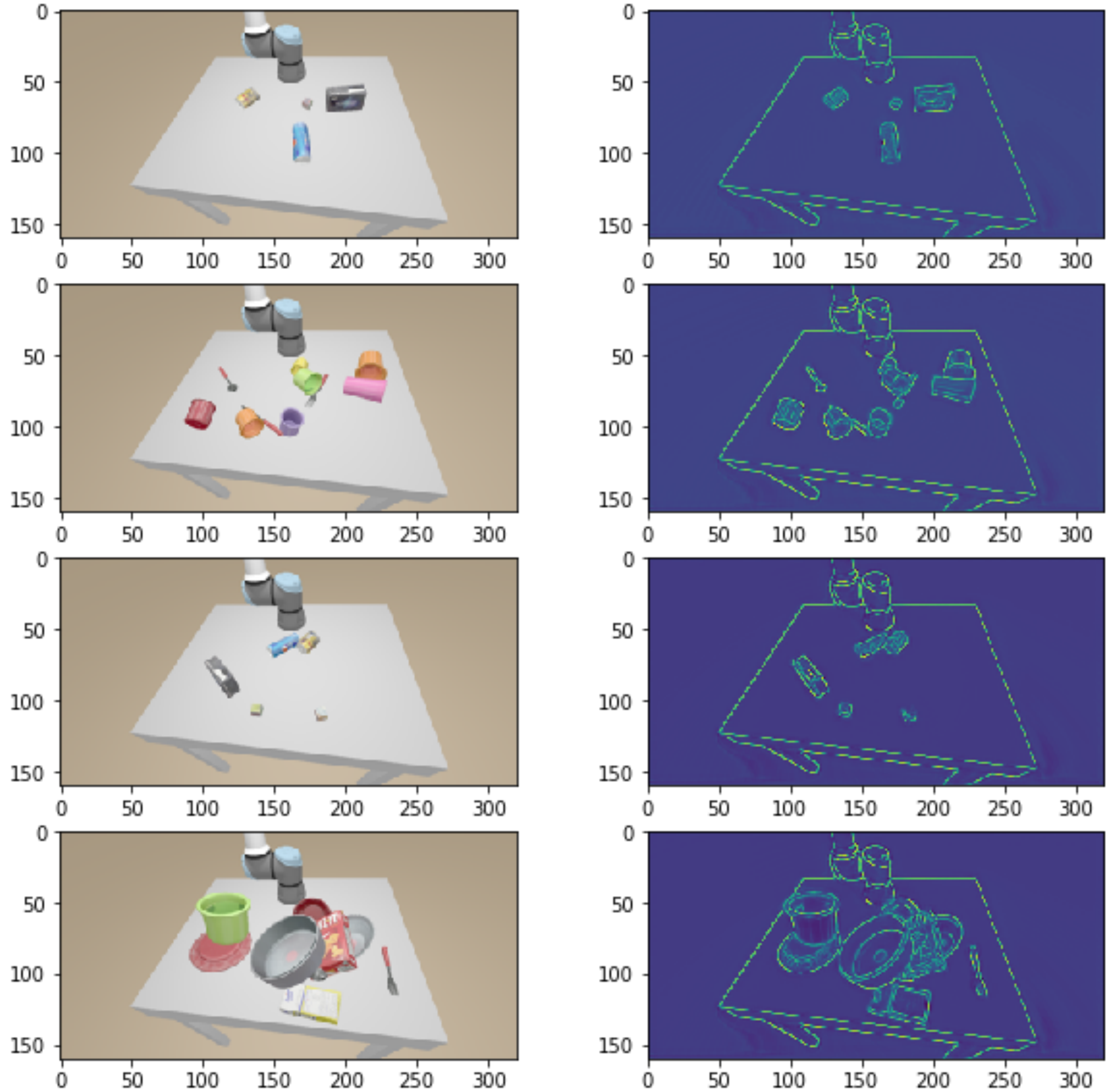
Out[86]:

```
<All keys matched successfully>
```

In [88]:
```
npz = np.load("test.npz")
test_images = npz["images"]

test_imgs= test_images.transpose((0, 3, 1, 2))
test_imgs = torch.tensor(test_imgs).float().contiguous()
test_imgs = test_imgs / 255
```

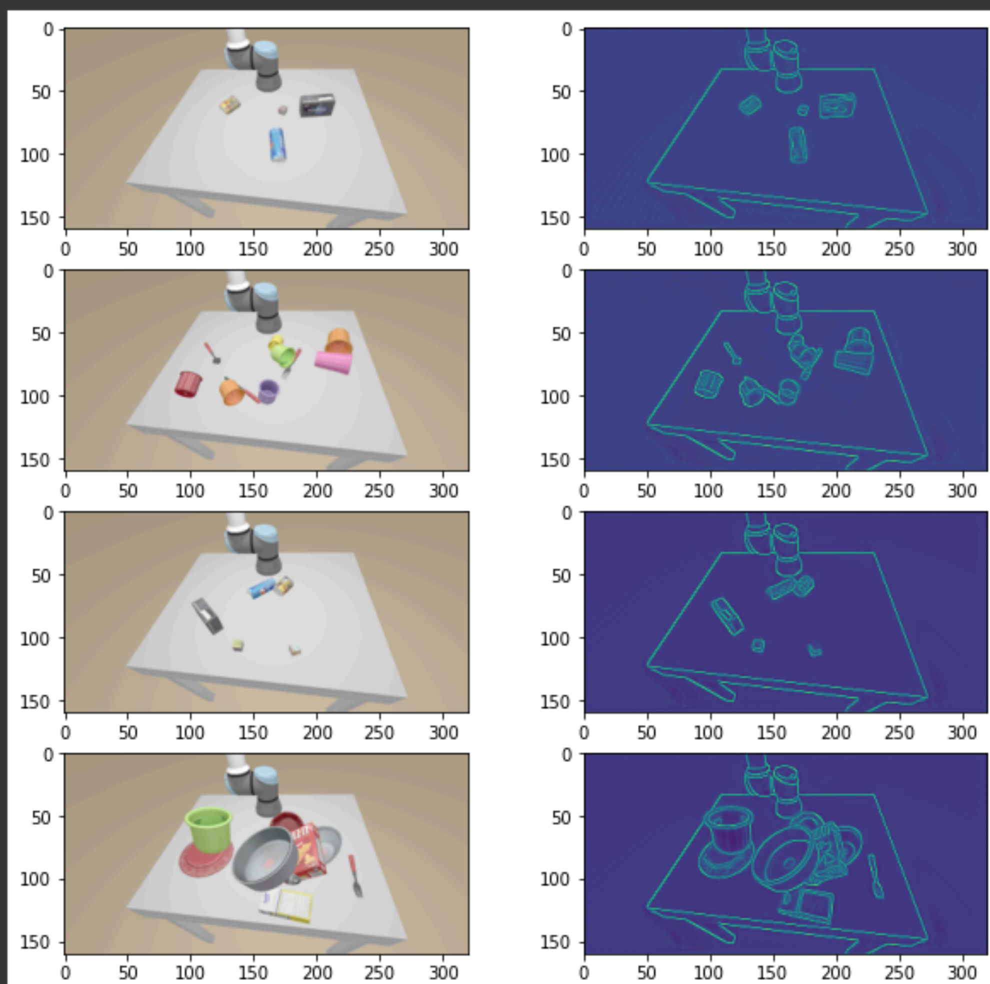
In [89]:
```
model.eval()
test = test_imgs.to(device)
pred = model(test).data.cpu().numpy()
```

In [90]:
```
plt.figure(figsize=(10, 10))
for i, img in enumerate(test_images[:4]):
    plt.subplot(4, 2, i * 2 + 1)
    plt.imshow(img)

    plt.subplot(4, 2, i * 2 + 2)
    # edge = evaluate your model on the test set, replace the followin
    edge = pred[i]
    edge = edge.reshape((160,320))
    plt.imshow(edge)
```



The orignal google colab trained result looks like this which is more smooth

```
plt.figure(figsize=(10, 10))
for i, img in enumerate(test_images[:4]):
    plt.subplot(4, 2, i * 2 + 1)
    plt.imshow(img)

    plt.subplot(4, 2, i * 2 + 2)
    # edge = evaluate your model on the test set, replace the following line
    edge = pred[i]
    edge = edge.reshape((160,320))
    plt.imshow(edge)
```



And attached is training statistics in google colab

```
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=25, gamma=0.1)

model = train_model(model, optimizer_ft, exp_lr_scheduler, num_epochs= 10)
```

```
Epoch 2/9
----------
LR 0.001
train: bce: 0.090844, loss: 0.090844
val: bce: 0.096931, loss: 0.096931
saving best model
0m 43s
Epoch 3/9
----------
LR 0.001
train: bce: 0.056724, loss: 0.056724
val: bce: 0.053438, loss: 0.053438
saving best model
0m 44s
Epoch 4/9
----------
LR 0.001
train: bce: 0.040137, loss: 0.040137
val: bce: 0.040042, loss: 0.040042
saving best model
0m 44s
Epoch 5/9
----------
LR 0.001
train: bce: 0.031046, loss: 0.031046
val: bce: 0.029555, loss: 0.029555
saving best model
0m 43s
Epoch 6/9
----------
LR 0.001
train: bce: 0.025532, loss: 0.025532
val: bce: 0.026134, loss: 0.026134
saving best model
0m 43s
Epoch 7/9
----------
LR 0.001
train: bce: 0.021659, loss: 0.021659
val: bce: 0.021111, loss: 0.021111
saving best model
0m 43s
Epoch 8/9
----------
LR 0.001
train: bce: 0.018970, loss: 0.018970
val: bce: 0.020706, loss: 0.020706
saving best model
0m 43s
Epoch 9/9
----------
LR 0.001
train: bce: 0.016971, loss: 0.016971
val: bce: 0.016993, loss: 0.016993
saving best model
0m 43s
Best val loss: 0.016993
```

In [ ]: