Image Segmentation by Supervised Learning

import libraries

In [1]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import numpy as np
import matplotlib.pyplot as plt
import math
from tqdm import tqdm
import random
import os
```

load data

In [2]:

```
directory_data  = './'
filename_data   = 'assignment_09_data.npz'
data            = np.load(os.path.join(directory_data, filename_data))

image_train     = data['image_train']
mask_train      = data['mask_train']

image_test      = data['image_test']
mask_test       = data['mask_test']

num_data_train  = image_train.shape[0]
num_data_test   = image_test.shape[0]
```

In [3]:

```
print('*************************************************')
print('size of x_train :', image_train.shape)
print('size of y_train :', mask_train.shape)
print('*************************************************')
print('size of x_test :', image_test.shape)
print('size of y_test :', mask_test.shape)
print('*************************************************')
```

```
*************************************************
size of x_train : (1000, 32, 32)
size of y_train : (1000, 32, 32)
*************************************************
size of x_test : (1000, 32, 32)
size of y_test : (1000, 32, 32)
*************************************************
```

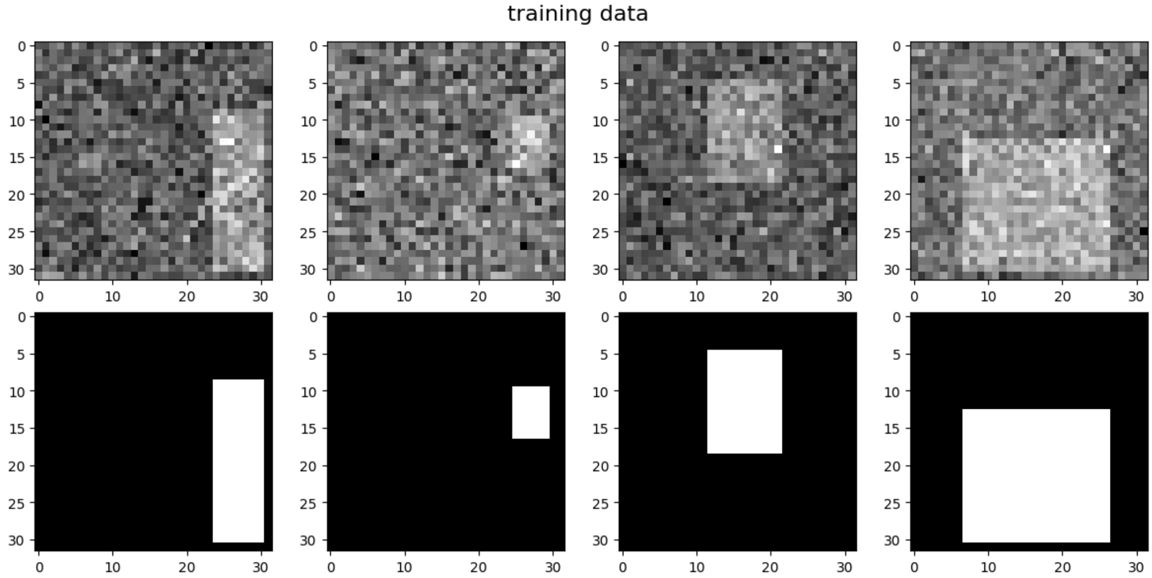
plot data

In [4]:

```
def plot_image(title, image, mask):

    nRow = 2
    nCol = 4
    size = 3

    fig, axes = plt.subplots(nRow, nCol, figsize=(size * nCol, size * nRow))
    fig.suptitle(title, fontsize=16)

    for c in range(nCol):
        axes[0, c].imshow(image[c], cmap='gray')
        axes[1, c].imshow(mask[c], cmap='gray', vmin=0, vmax=1)

    plt.tight_layout()
    plt.show()
```

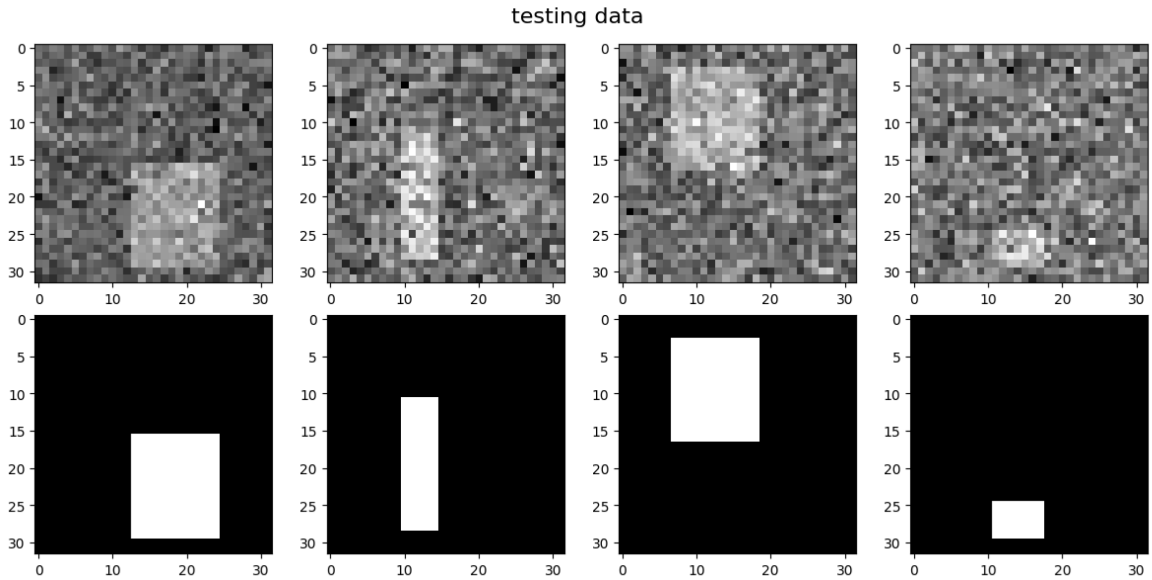
In [5]:

```
plot_image('training data', image_train, mask_train)
```



In [6]:

```
plot_image('testing data', image_test, mask_test)
```



custom data loader for the PyTorch framework

In [7]:

```
class dataset(Dataset):

    def __init__(self, image, mask, use_transform=False):

        self.image           = image
        self.mask            = mask
        self.use_transform   = use_transform

    def __getitem__(self, index):

        image    = self.image[index]
        mask     = self.mask[index]

        image    = torch.FloatTensor(image).unsqueeze(dim=0)
        mask     = torch.FloatTensor(mask).unsqueeze(dim=0)

#         image    = (image - torch.min(image)) / (torch.max(image) - torch.min(i
mage))

        if self.use_transform:
            # ==================================================
            # add codes for applying data augmentation
            #

            self.transform = transforms.RandomChoice([
#                 transforms.RandomResizedCrop(size=32),
                transforms.RandomVerticalFlip(),
                transforms.RandomHorizontalFlip(),
#                 transforms.RandomAffine(degrees=90, scale=(0.5, 1.5), shear=(4
5,45,45,45)),
                transforms.RandomRotation(180),
#                 transforms.Normalize([0.5],[0.5]),
            ])

            seed = np.random.randint(20184757)
            random.seed(seed)
#             np.random.seed(seed)
            torch.manual_seed(seed)

            image    = self.transform(image)

            random.seed(seed)
#             np.random.seed(seed)
            torch.manual_seed(seed)

            mask     = self.transform(mask)

            #
            # ==================================================

        return (image, mask)

    def __len__(self):

        number_image = self.image.shape[0]

        return number_image
```

setting device

In [8]:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'mps')
```

In [9]:

```
print(device)
```

```
mps
```

In [10]:

```
# random seed
random.seed(20184757)
np.random.seed(20184757)
torch.manual_seed(20184757)
torch.cuda.manual_seed(20184757)
torch.cuda.manual_seed_all(20184757)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

construct datasets and dataloaders for testing and testing

In [11]:

```
# ==================================================
# determine the mini-batch size
#
size_minibatch      = 32
#
# ==================================================

dataset_train       = dataset(image_train, mask_train, True)
dataset_test        = dataset(image_test, mask_test, False)

dataloader_train    = torch.utils.data.DataLoader(dataset_train, batch_size=size
_minibatch, shuffle=True, drop_last=True)
dataloader_test     = torch.utils.data.DataLoader(dataset_test, batch_size=size_
minibatch, shuffle=False, drop_last=False)
```

construct a neural network

In [12]:

```
class Network(nn.Module):
    def __init__(self):
        super(Network,self).__init__()

        # -------------------------------------------------
        # Encoder
        # -------------------------------------------------
        self.encoder_layer1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=64, kernel_size=3, stride=1, p
adding=1, bias=True),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1,
padding=1, bias=True),
            nn.BatchNorm2d(64),
            nn.ReLU(),
        )

#         self.down_layer1 = nn.Sequential(
#             nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, stride=
2, padding=1, bias=True),
#             nn.BatchNorm2d(32),
#             nn.ReLU()
#         )

        self.encoder_layer2 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1,
padding=1, bias=True),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1
, padding=1, bias=True),
            nn.BatchNorm2d(128),
            nn.ReLU(),
        )

#         self.down_layer2 = nn.Sequential(
#             nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=
2, padding=1, bias=True),
#             nn.BatchNorm2d(64),
#             nn.ReLU()
#         )

        self.encoder_layer3 = nn.Sequential(
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=1
, padding=1, bias=True),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1
, padding=1, bias=True),
            nn.BatchNorm2d(256),
            nn.ReLU(),
        )

        self.encoder_layer4 = nn.Sequential(
            nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, stride=1
, padding=1, bias=True),
```

```
            nn.BatchNorm2d(512),
            nn.ReLU(),
        )

        self.down_layer = nn.Sequential(
            nn.MaxPool2d((2,2)),
        )

        # ------------------------------------------------
        # Decoder
        # ------------------------------------------------
#         self.up_layer = nn.Sequential(
#             nn.Upsample(scale_factor=2, mode='bilinear', align_corners=False),
#         )

        self.decoder_layer4 = nn.Sequential(
            nn.Conv2d(in_channels=512, out_channels=512, kernel_size=3, stride=1
, padding=1, bias=True),
            nn.BatchNorm2d(512),
            nn.ReLU(),
        )

        self.up_layer3 = nn.Sequential(
            nn.ConvTranspose2d(in_channels=512, out_channels=256, kernel_size=2,
stride=2, bias=True)
        )

        self.decoder_layer3 = nn.Sequential(
            nn.Conv2d(in_channels=512, out_channels=256, kernel_size=3, stride=1
, padding=1, bias=True),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, stride=1
, padding=1, bias=True),
            nn.BatchNorm2d(256),
            nn.ReLU(),
        )

        self.up_layer2 = nn.Sequential(
            nn.ConvTranspose2d(in_channels=256, out_channels=128, kernel_size=2,
stride=2, bias=True),
#             nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=
3, stride=2, padding=1, bias=True, output_padding=1),
#             nn.BatchNorm2d(64),
#             nn.ReLU(),
        )

        self.decoder_layer2 = nn.Sequential(
            nn.Conv2d(in_channels=256, out_channels=128, kernel_size=3, stride=1
, padding=1, bias=True),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, stride=1
, padding=1, bias=True),
            nn.BatchNorm2d(128),
            nn.ReLU(),
        )

        self.up_layer1 = nn.Sequential(
```

```
            nn.ConvTranspose2d(in_channels=128, out_channels=64, kernel_size=2,
stride=2, bias=True)
#               nn.ConvTranspose2d(in_channels=64, out_channels=32, kernel_size=3,
stride=2, padding=1, bias=True, output_padding=1),
#               nn.BatchNorm2d(32),
#               nn.ReLU(),
        )

        self.decoder_layer1 = nn.Sequential(
            nn.Conv2d(in_channels=128, out_channels=64, kernel_size=3, stride=1,
padding=1, bias=True),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=1, kernel_size=3, stride=1, p
adding=1, bias=True),
            nn.Sigmoid(),
        )

        # --------------------------------------------------
        # Network
        # --------------------------------------------------
#         self.network = nn.Sequential(
#             self.encoder_layer1,
#             self.encoder_layer2,
#             self.decoder_layer2,
#             self.decoder_layer1,
#         )

        self.network = nn.ModuleList([
            self.encoder_layer1,
            self.encoder_layer2,
            self.encoder_layer3,
            self.encoder_layer4,
            self.down_layer,
#             self.down_layer1,
#             self.down_layer2,
            self.up_layer3,
            self.up_layer2,
            self.up_layer1,
            self.decoder_layer4,
            self.decoder_layer3,
            self.decoder_layer2,
            self.decoder_layer1,
        ])

        self.initialize_weight()

    def forward(self,x):

#         out = self.network(x)

        encode1 = self.encoder_layer1(x)
#         down1 = self.down_layer1(encode1)
        down1 = self.down_layer(encode1)

        encode2 = self.encoder_layer2(down1)
#         down2 = self.down_layer2(encode2)
        down2 = self.down_layer(encode2)
```

```
        encode3 = self.encoder_layer3(down2)
        down3 = self.down_layer(encode3)

        encode4 = self.encoder_layer4(down3)

        decode4 = self.decoder_layer4(encode4)

        up3 = self.up_layer3(decode4)
        cat3 = torch.cat((up3, encode3), dim=1)
        decode3 = self.decoder_layer3(cat3)

        up2 = self.up_layer2(decode3)
        cat2 = torch.cat((up2, encode2), dim=1)
        decode2 = self.decoder_layer2(cat2)

        up1 = self.up_layer1(decode2)
        cat1 = torch.cat((up1, encode1), dim=1)
        out = self.decoder_layer1(cat1)

        return out

    # ==========================================================================
    # initialize weights
    # ==========================================================================
    def initialize_weight(self):

        for m in self.network.modules():

            if isinstance(m, nn.Conv2d):

                nn.init.xavier_uniform_(m.weight)
                if m.bias is not None:

                    nn.init.constant_(m.bias, 1)
                    pass

            elif isinstance(m, nn.BatchNorm2d):

                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 1)

            elif isinstance(m, nn.Linear):

                nn.init.xavier_uniform_(m.weight)

                if m.bias is not None:

                    nn.init.constant_(m.bias, 1)
                    pass
```

build the network

In [13]:

```
model        = Network().to(device)

# ==================================================
# determine the optimiser and its associated hyper-parameters
#
learning_rate    = 0.0001
alpha            = 0.0001
weight_decay     = 0.001
number_epoch     = 493
optimizer        = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)
#
# ==================================================
```

compute the prediction

In [14]:

```
def compute_prediction(model, input):

    prediction = model(input)

    return prediction
```

compute the loss

In [15]:

```
def compute_loss_data_fidelity(prediction, mask):
    # ==================================================
    # fill up the blank
    #

    # nn.BCEWithLogitsLoss
    # -> combines Sigmoid layer and BCELoss in one single class
    # -> more numerically stable than using plain Sigmoid followed by BCELoss
    # => can't use due to regularization loss
    loss = nn.BCELoss()(prediction, mask)

    #
    # ==================================================

    return loss
```

In [16]:

```
def compute_loss_regularization(prediction):
    # ==================================================
    # fill up the blank
    #

    loss = (torch.sum(torch.abs(torch.sub(prediction[:,:,1:,:], prediction[:,:,:
-1,:]))) \
            + torch.sum(torch.abs(torch.sub(prediction[:,:,:,1:], prediction
[:,:,:,:-1])))) \
    / (prediction.size(0) * prediction.size(1) * prediction.size(2) * prediction
.size(3))

    #
    # ==================================================

    return loss
```

In [17]:

```
def compute_loss(prediction, mask, alpha):
    # ==================================================
    # fill up the blank
    #

    loss_data_fidelity  = compute_loss_data_fidelity(prediction, mask)
    loss_regularization = compute_loss_regularization(prediction)
    loss                = loss_data_fidelity + alpha * loss_regularization

    #
    # ==================================================

    return (loss, loss_data_fidelity, loss_regularization)
```

compute the loss value

In [18]:

```
def compute_loss_value(loss):

    loss_value = loss.item()

    return loss_value
```

compute the accuracy

In [19]:

```
def compute_accuracy(prediction, mask):

    prediction   = prediction.squeeze(axis=1)
    binary       = (prediction >= 0.5)
    mask         = mask.squeeze(axis=1).bool()

    intersection = (binary & mask).float().sum((1, 2))
    union         = (binary | mask).float().sum((1, 2))

    eps          = 1e-8
    correct      = (intersection + eps) / (union + eps)
    accuracy     = correct.mean() * 100.0
    accuracy     = accuracy.cpu()

    return accuracy
```

Variable for the learning curves

In [20]:

```
loss_train_mean     = np.zeros(number_epoch)
loss_train_std      = np.zeros(number_epoch)
accuracy_train_mean = np.zeros(number_epoch)
accuracy_train_std  = np.zeros(number_epoch)

loss_test_mean      = np.zeros(number_epoch)
loss_test_std       = np.zeros(number_epoch)
accuracy_test_mean  = np.zeros(number_epoch)
accuracy_test_std   = np.zeros(number_epoch)

loss_train_data_fidelity_mean   = np.zeros(number_epoch)
loss_train_data_fidelity_std    = np.zeros(number_epoch)
loss_train_regularization_mean  = np.zeros(number_epoch)
loss_train_regularization_std   = np.zeros(number_epoch)

loss_test_data_fidelity_mean    = np.zeros(number_epoch)
loss_test_data_fidelity_std     = np.zeros(number_epoch)
loss_test_regularization_mean   = np.zeros(number_epoch)
loss_test_regularization_std    = np.zeros(number_epoch)
```

train

In [21]:

```
def train(model, optimizer, dataloader):

    loss_epoch                    = []
    loss_data_fidelity_epoch      = []
    loss_regularization_epoch     = []
    accuracy_epoch                = []

    model.train()

    for index_batch, (image, mask) in enumerate(dataloader):

        image       = image.to(device)
        mask        = mask.to(device)

        # ==================================================
        # fill up the blank
        #
        prediction  = compute_prediction(model, image)
        (loss, loss_data_fidelity, loss_regularization) = compute_loss(predictio
n, mask, alpha)

        loss_value                  = compute_loss_value(loss)
        loss_data_fidelity_value    = compute_loss_value(loss_data_fidelity)
        loss_regularization_value   = compute_loss_value(loss_regularization)
        accuracy                    = compute_accuracy(prediction, mask)
        #
        # ==================================================

        loss_epoch.append(loss_value)
        loss_data_fidelity_epoch.append(loss_data_fidelity_value)
        loss_regularization_epoch.append(loss_regularization_value)
        accuracy_epoch.append(accuracy)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    loss_mean    = np.mean(loss_epoch)
    loss_std     = np.std(loss_epoch)

    loss_data_fidelity_mean = np.mean(loss_data_fidelity_epoch)
    loss_data_fidelity_std  = np.std(loss_data_fidelity_epoch)

    loss_regularization_mean     = np.mean(loss_regularization_epoch)
    loss_regularization_std      = np.std(loss_regularization_epoch)

    accuracy_mean    = np.mean(accuracy_epoch)
    accuracy_std     = np.std(accuracy_epoch)

    loss                = {'mean' : loss_mean, 'std' : loss_std}
    loss_data_fidelity  = {'mean' : loss_data_fidelity_mean, 'std' : loss_data_f
idelity_std}
    loss_regularization = {'mean' : loss_regularization_mean, 'std' : loss_regul
arization_std}
    accuracy            = {'mean' : accuracy_mean, 'std' : accuracy_std}

    return (loss, loss_data_fidelity, loss_regularization, accuracy)
```

test

In [22]:

```
def test(model, dataloader):

    loss_epoch                      = []
    loss_data_fidelity_epoch        = []
    loss_regularization_epoch       = []
    accuracy_epoch                  = []

    model.eval()

    for index_batch, (image, mask) in enumerate(dataloader):

        image       = image.to(device)
        mask        = mask.to(device)

        # ==================================================
        # fill up the blank
        #
        prediction  = compute_prediction(model, image)
        (loss, loss_data_fidelity, loss_regularization) = compute_loss(predictio
n, mask, alpha)

        loss_value                  = compute_loss_value(loss)
        loss_data_fidelity_value    = compute_loss_value(loss_data_fidelity)
        loss_regularization_value   = compute_loss_value(loss_regularization)
        accuracy                    = compute_accuracy(prediction, mask)
        #
        # ==================================================

        loss_epoch.append(loss_value)
        loss_data_fidelity_epoch.append(loss_data_fidelity_value)
        loss_regularization_epoch.append(loss_regularization_value)
        accuracy_epoch.append(accuracy)

    loss_mean   = np.mean(loss_epoch)
    loss_std    = np.std(loss_epoch)

    loss_data_fidelity_mean = np.mean(loss_data_fidelity_epoch)
    loss_data_fidelity_std  = np.std(loss_data_fidelity_epoch)

    loss_regularization_mean    = np.mean(loss_regularization_epoch)
    loss_regularization_std     = np.std(loss_regularization_epoch)

    accuracy_mean   = np.mean(accuracy_epoch)
    accuracy_std    = np.std(accuracy_epoch)

    loss                = {'mean' : loss_mean, 'std' : loss_std}
    loss_data_fidelity  = {'mean' : loss_data_fidelity_mean, 'std' : loss_data_f
idelity_std}
    loss_regularization = {'mean' : loss_regularization_mean, 'std' : loss_regul
arization_std}
    accuracy            = {'mean' : accuracy_mean, 'std' : accuracy_std}

    return (loss, loss_data_fidelity, loss_regularization, accuracy)
```

train and test

In [23]:

```
# ================================================================================
==
#
# iterations for epochs
#
# ================================================================================
==
for i in tqdm(range(number_epoch)):
    # ===========================================================================
======
    #
    # training
    #
    # ===========================================================================
======
    (loss_train, loss_data_fidelity_train, loss_regularization_train, accuracy_t
rain) = train(model, optimizer, dataloader_train)

    loss_train_mean[i]  = loss_train['mean']
    loss_train_std[i]   = loss_train['std']

    loss_train_data_fidelity_mean[i]    = loss_data_fidelity_train['mean']
    loss_train_data_fidelity_std[i]     = loss_data_fidelity_train['std']

    loss_train_regularization_mean[i]   = loss_regularization_train['mean']
    loss_train_regularization_std[i]    = loss_regularization_train['std']

    accuracy_train_mean[i]  = accuracy_train['mean']
    accuracy_train_std[i]   = accuracy_train['std']

    # ===========================================================================
======
    #
    # testing
    #
    # ===========================================================================
======
    (loss_test, loss_data_fidelity_test, loss_regularization_test, accuracy_test
) = test(model, dataloader_test)

    loss_test_mean[i]  = loss_test['mean']
    loss_test_std[i]   = loss_test['std']

    loss_test_data_fidelity_mean[i]    = loss_data_fidelity_test['mean']
    loss_test_data_fidelity_std[i]     = loss_data_fidelity_test['std']

    loss_test_regularization_mean[i]   = loss_regularization_test['mean']
    loss_test_regularization_std[i]    = loss_regularization_test['std']

    accuracy_test_mean[i]  = accuracy_test['mean']
    accuracy_test_std[i]   = accuracy_test['std']
```

```
  0%|                                                    | 0/493 [00:
00<?, ?it/s]/var/folders/qy/qpwfmy2x5kz6jkcq1rb246x80000gn/T/ipykern
el_71957/2959453998.py:7: UserWarning: The operator 'aten::bitwise_a
nd.Tensor_out' is not currently supported on the MPS backend and wil
l fall back to run on the CPU. This may have performance implication
s. (Triggered internally at  /Users/runner/work/pytorch/pytorch/pyto
rch/aten/src/ATen/mps/MPSFallback.mm:11.)
  intersection = (binary & mask).float().sum((1, 2))
100%|██████████████████████████████████████████| 493/493 [58:28<00:0
0,  7.12s/it]
```

---

functions for presenting the results

In [24]:

```
def function_result_01():

    title           = 'loss (training)'
    label_axis_x    = 'epoch'
    label_axis_y    = 'loss'
    color_mean      = 'red'
    color_std       = 'blue'
    alpha           = 0.3

    plt.figure(figsize=(8, 6))
    plt.title(title)

    plt.plot(range(len(loss_train_mean)), loss_train_mean, '-', color = color_me
an)
    plt.fill_between(range(len(loss_train_mean)), loss_train_mean - loss_train_s
td, loss_train_mean + loss_train_std, facecolor = color_std, alpha = alpha)

    plt.xlabel(label_axis_x)
    plt.ylabel(label_axis_y)

    plt.tight_layout()
    plt.show()
```

In [25]:

```
def function_result_02():

    title           = 'loss - data fidelity (training)'
    label_axis_x    = 'epoch'
    label_axis_y    = 'loss'
    color_mean      = 'red'
    color_std       = 'blue'
    alpha           = 0.3

    plt.figure(figsize=(8, 6))
    plt.title(title)

    plt.plot(range(len(loss_train_data_fidelity_mean)), loss_train_data_fidelity
_mean, '-', color = color_mean)
    plt.fill_between(range(len(loss_train_data_fidelity_mean)), loss_train_data_
fidelity_mean - loss_train_data_fidelity_std, loss_train_data_fidelity_mean + lo
ss_train_data_fidelity_std, facecolor = color_std, alpha = alpha)

    plt.xlabel(label_axis_x)
    plt.ylabel(label_axis_y)

    plt.tight_layout()
    plt.show()
```

In [26]:

```
def function_result_03():

    title           = 'loss - regularization (training)'
    label_axis_x    = 'epoch'
    label_axis_y    = 'loss'
    color_mean      = 'red'
    color_std       = 'blue'
    alpha           = 0.3

    plt.figure(figsize=(8, 6))
    plt.title(title)

    plt.plot(range(len(loss_train_regularization_mean)), loss_train_regularizati
on_mean, '-', color = color_mean)
    plt.fill_between(range(len(loss_train_regularization_mean)), loss_train_regu
larization_mean - loss_train_regularization_std, loss_train_regularization_mean
+ loss_train_regularization_std, facecolor = color_std, alpha = alpha)

    plt.xlabel(label_axis_x)
    plt.ylabel(label_axis_y)

    plt.tight_layout()
    plt.show()
```

In [27]:

```
def function_result_04():

    title           = 'loss (testing)'
    label_axis_x    = 'epoch'
    label_axis_y    = 'loss'
    color_mean      = 'red'
    color_std       = 'blue'
    alpha           = 0.3

    plt.figure(figsize=(8, 6))
    plt.title(title)

    plt.plot(range(len(loss_test_mean)), loss_test_mean, '-', color = color_mean
)
    plt.fill_between(range(len(loss_test_mean)), loss_test_mean - loss_test_std,
loss_test_mean + loss_test_std, facecolor = color_std, alpha = alpha)

    plt.xlabel(label_axis_x)
    plt.ylabel(label_axis_y)

    plt.tight_layout()
    plt.show()
```

In [28]:

```
def function_result_05():

    title           = 'loss - data fidelity (testing)'
    label_axis_x    = 'epoch'
    label_axis_y    = 'loss'
    color_mean      = 'red'
    color_std       = 'blue'
    alpha           = 0.3

    plt.figure(figsize=(8, 6))
    plt.title(title)

    plt.plot(range(len(loss_test_data_fidelity_mean)), loss_test_data_fidelity_m
ean, '-', color = color_mean)
    plt.fill_between(range(len(loss_test_data_fidelity_mean)), loss_test_data_fi
delity_mean - loss_test_data_fidelity_std, loss_test_data_fidelity_mean + loss_t
est_data_fidelity_std, facecolor = color_std, alpha = alpha)

    plt.xlabel(label_axis_x)
    plt.ylabel(label_axis_y)

    plt.tight_layout()
    plt.show()
```

In [29]:

```
def function_result_06():

    title               = 'loss - regularization (testing)'
    label_axis_x        = 'epoch'
    label_axis_y        = 'loss'
    color_mean          = 'red'
    color_std           = 'blue'
    alpha               = 0.3

    plt.figure(figsize=(8, 6))
    plt.title(title)

    plt.plot(range(len(loss_test_regularization_mean)), loss_test_regularization
_mean, '-', color = color_mean)
    plt.fill_between(range(len(loss_test_regularization_mean)), loss_test_regula
rization_mean - loss_test_regularization_std, loss_test_regularization_mean + lo
ss_test_regularization_std, facecolor = color_std, alpha = alpha)

    plt.xlabel(label_axis_x)
    plt.ylabel(label_axis_y)

    plt.tight_layout()
    plt.show()
```

In [30]:

```
def function_result_07():

    title               = 'accuracy (training)'
    label_axis_x        = 'epoch'
    label_axis_y        = 'accuracy'
    color_mean          = 'red'
    color_std           = 'blue'
    alpha               = 0.3

    plt.figure(figsize=(8, 6))
    plt.title(title)

    plt.plot(range(len(accuracy_train_mean)), accuracy_train_mean, '-', color =
color_mean)
    plt.fill_between(range(len(accuracy_train_mean)), accuracy_train_mean - accu
racy_train_std, accuracy_train_mean + accuracy_train_std, facecolor = color_std,
alpha = alpha)

    plt.xlabel(label_axis_x)
    plt.ylabel(label_axis_y)

    plt.tight_layout()
    plt.show()
```

In [31]:

```
def function_result_08():

    title           = 'accuracy (testing)'
    label_axis_x    = 'epoch'
    label_axis_y    = 'accuracy'
    color_mean      = 'red'
    color_std       = 'blue'
    alpha           = 0.3

    plt.figure(figsize=(8, 6))
    plt.title(title)

    plt.plot(range(len(accuracy_test_mean)), accuracy_test_mean, '-', color = co
lor_mean)
    plt.fill_between(range(len(accuracy_test_mean)), accuracy_test_mean - accura
cy_test_std, accuracy_test_mean + accuracy_test_std, facecolor = color_std, alph
a = alpha)

    plt.xlabel(label_axis_x)
    plt.ylabel(label_axis_y)

    plt.tight_layout()
    plt.show()
```

In [32]:

```
def function_result_09():

    nRow = 10
    nCol = 4
    size = 3

    title = 'training results'
    fig, axes = plt.subplots(nRow, nCol, figsize=(size * nCol, size * nRow))
    fig.suptitle(title, fontsize=16)

    number_data = len(dataset_train)
    index_image = np.linspace(0, number_data-1, nRow).astype(int)

    image        = torch.FloatTensor(dataset_train.image[index_image]).unsqueeze(
dim=1).to(device)
    mask         = torch.FloatTensor(dataset_train.mask[index_image]).unsqueeze(d
im=1).to(device)
    prediction   = compute_prediction(model, image)

    image        = image.detach().cpu().squeeze(axis=1)
    mask         = mask.detach().cpu().squeeze(axis=1)
    prediction   = prediction.detach().cpu().squeeze(axis=1)
    binary       = (prediction >= 0.5)

    for r in range(nRow):

            axes[r, 0].imshow(image[r], cmap='gray')
            axes[r, 1].imshow(prediction[r], cmap='gray', vmin=0, vmax=1)
            axes[r, 2].imshow(binary[r], cmap='gray', vmin=0, vmax=1)
            axes[r, 3].imshow(mask[r], cmap='gray', vmin=0, vmax=1)

            axes[r, 0].xaxis.set_visible(False)
            axes[r, 1].xaxis.set_visible(False)
            axes[r, 2].xaxis.set_visible(False)
            axes[r, 3].xaxis.set_visible(False)

            axes[r, 0].yaxis.set_visible(False)
            axes[r, 1].yaxis.set_visible(False)
            axes[r, 2].yaxis.set_visible(False)
            axes[r, 3].yaxis.set_visible(False)

    plt.tight_layout()
    plt.show()
```

In [33]:

```
def function_result_10():

    nRow = 10
    nCol = 4
    size = 3

    title = 'testing results'
    fig, axes = plt.subplots(nRow, nCol, figsize=(size * nCol, size * nRow))
    fig.suptitle(title, fontsize=16)

    number_data = len(dataset_test)
    index_image = np.linspace(0, number_data-1, nRow).astype(int)

    image       = torch.FloatTensor(dataset_test.image[index_image]).unsqueeze(d
im=1).to(device)
    mask        = torch.FloatTensor(dataset_test.mask[index_image]).unsqueeze(di
m=1).to(device)
    prediction  = compute_prediction(model, image)

    image       = image.detach().cpu().squeeze(axis=1)
    mask        = mask.detach().cpu().squeeze(axis=1)
    prediction  = prediction.detach().cpu().squeeze(axis=1)
    binary      = (prediction >= 0.5)

    for r in range(nRow):

            axes[r, 0].imshow(image[r], cmap='gray')
            axes[r, 1].imshow(prediction[r], cmap='gray', vmin=0, vmax=1)
            axes[r, 2].imshow(binary[r], cmap='gray', vmin=0, vmax=1)
            axes[r, 3].imshow(mask[r], cmap='gray', vmin=0, vmax=1)

            axes[r, 0].xaxis.set_visible(False)
            axes[r, 1].xaxis.set_visible(False)
            axes[r, 2].xaxis.set_visible(False)
            axes[r, 3].xaxis.set_visible(False)

            axes[r, 0].yaxis.set_visible(False)
            axes[r, 1].yaxis.set_visible(False)
            axes[r, 2].yaxis.set_visible(False)
            axes[r, 3].yaxis.set_visible(False)

    plt.tight_layout()
    plt.show()
```

In [34]:

```
def function_result_11():

    print('final training accuracy = %9.8f' % (accuracy_train_mean[-1]))
```

In [35]:

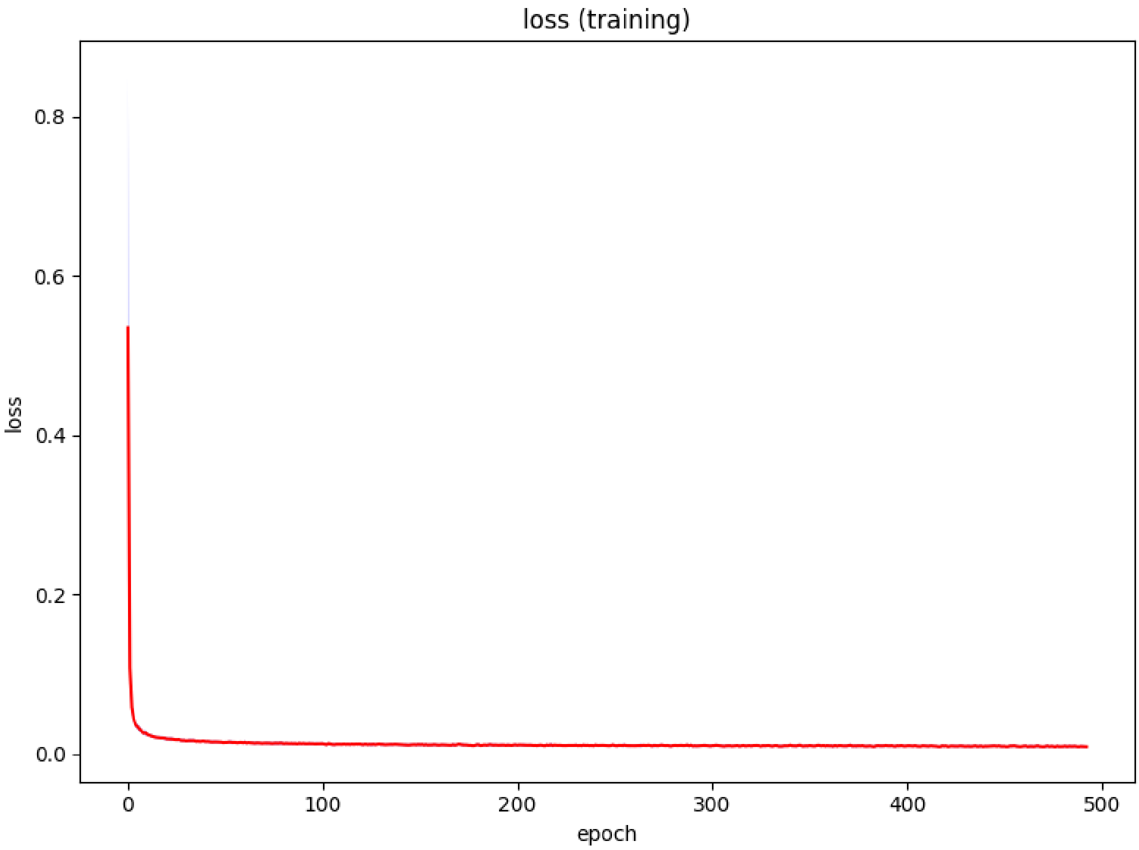
```
def function_result_12():

    print('final testing accuracy = %9.8f' % (accuracy_test_mean[-1]))
```

results

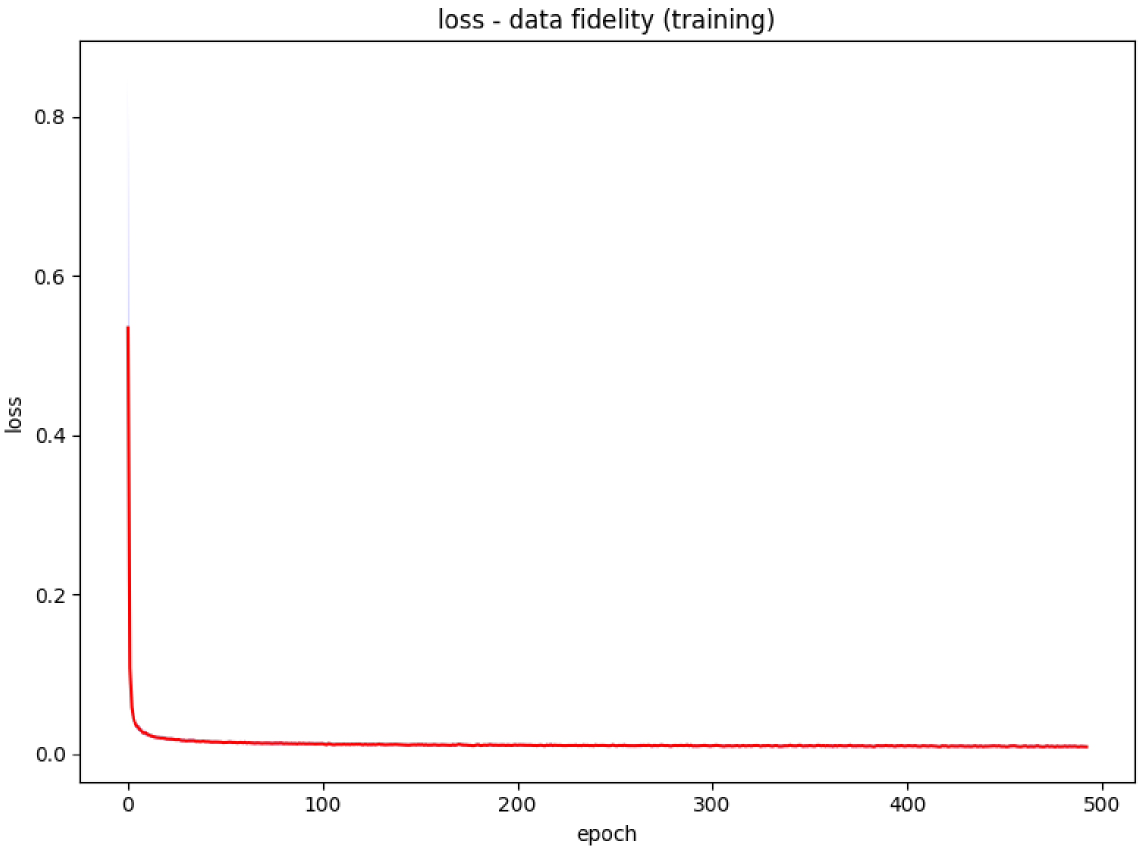
In [36]:

```
number_result = 12

for i in range(number_result):

    title           = '# RESULT # {:02d}'.format(i+1)
    name_function   = 'function_result_{:02d}()'.format(i+1)

    print('')
    print('######################################################################
###########')
    print('#')
    print(title)
    print('#')
    print('######################################################################
###########')
    print('')

    eval(name_function)
```

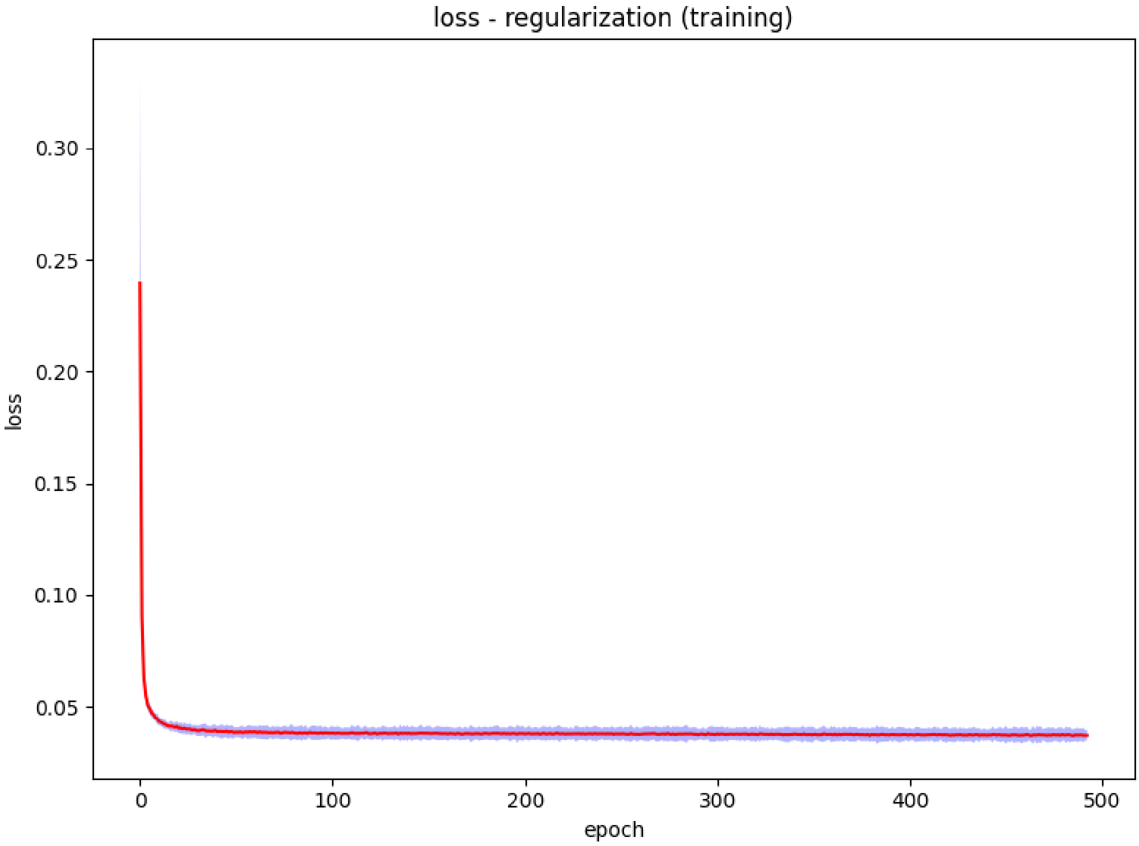
```
########################################################################
############
#
# RESULT # 01
#
########################################################################
############
```



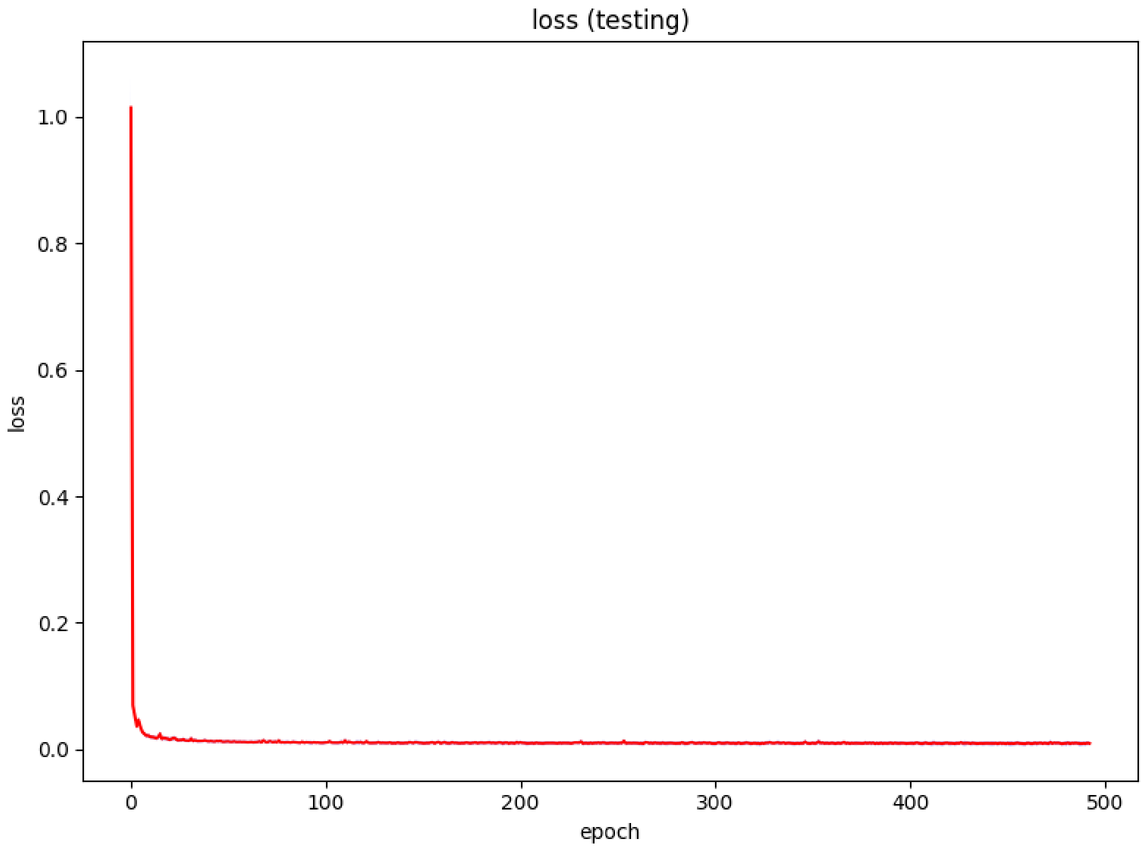
```
####################################################################
############
#
# RESULT # 02
#
####################################################################
############
```



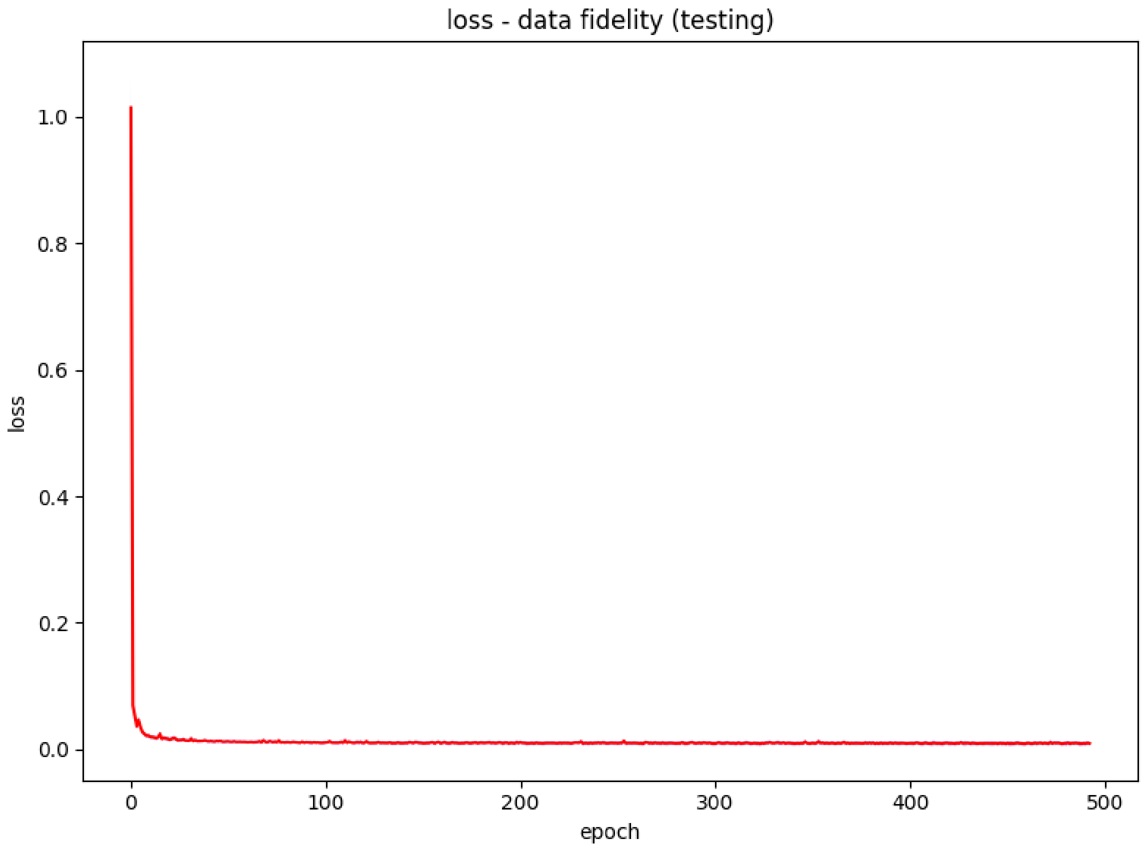

```
####################################################################
############
#
# RESULT # 03
#
####################################################################
############
```


loss - regularization (training)
loss
epoch
0.30
0.25
0.20
0.15
0.10
0.05
0
100
200
300
400
500

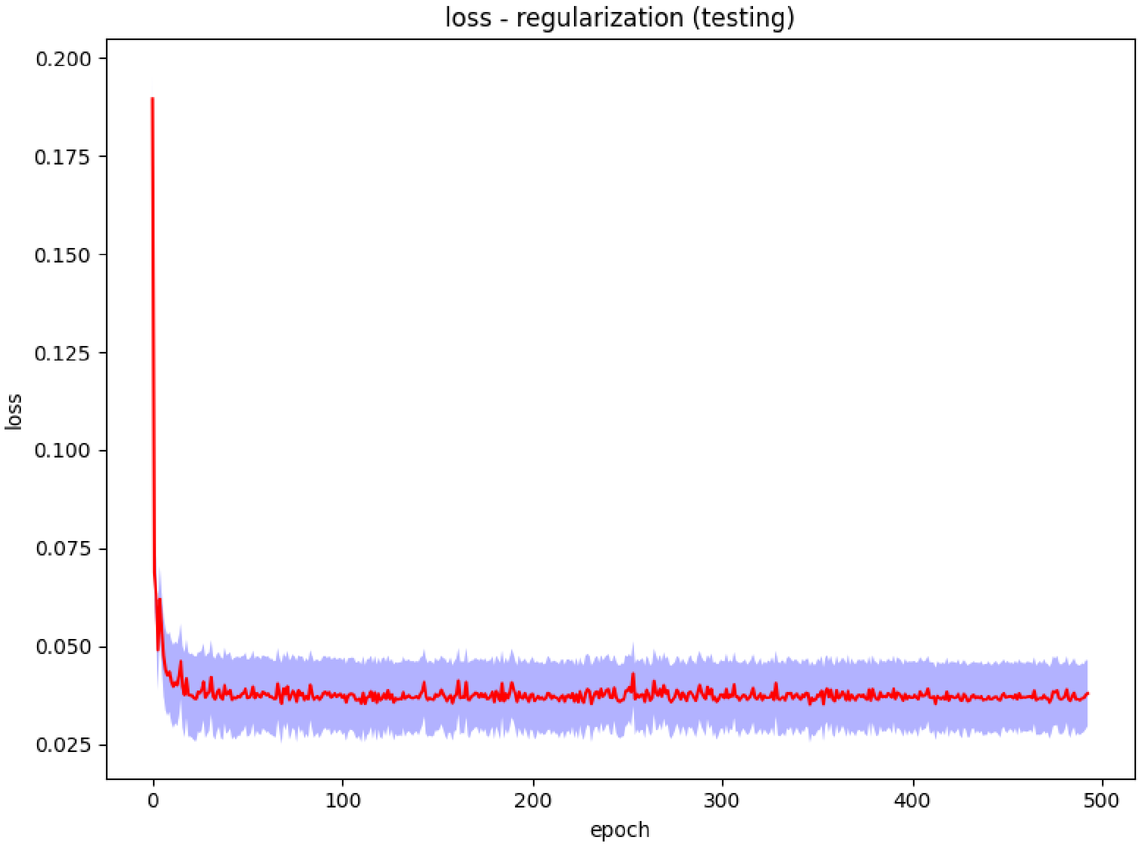
```
########################################################################
############
#
# RESULT # 04
#
########################################################################
############
```



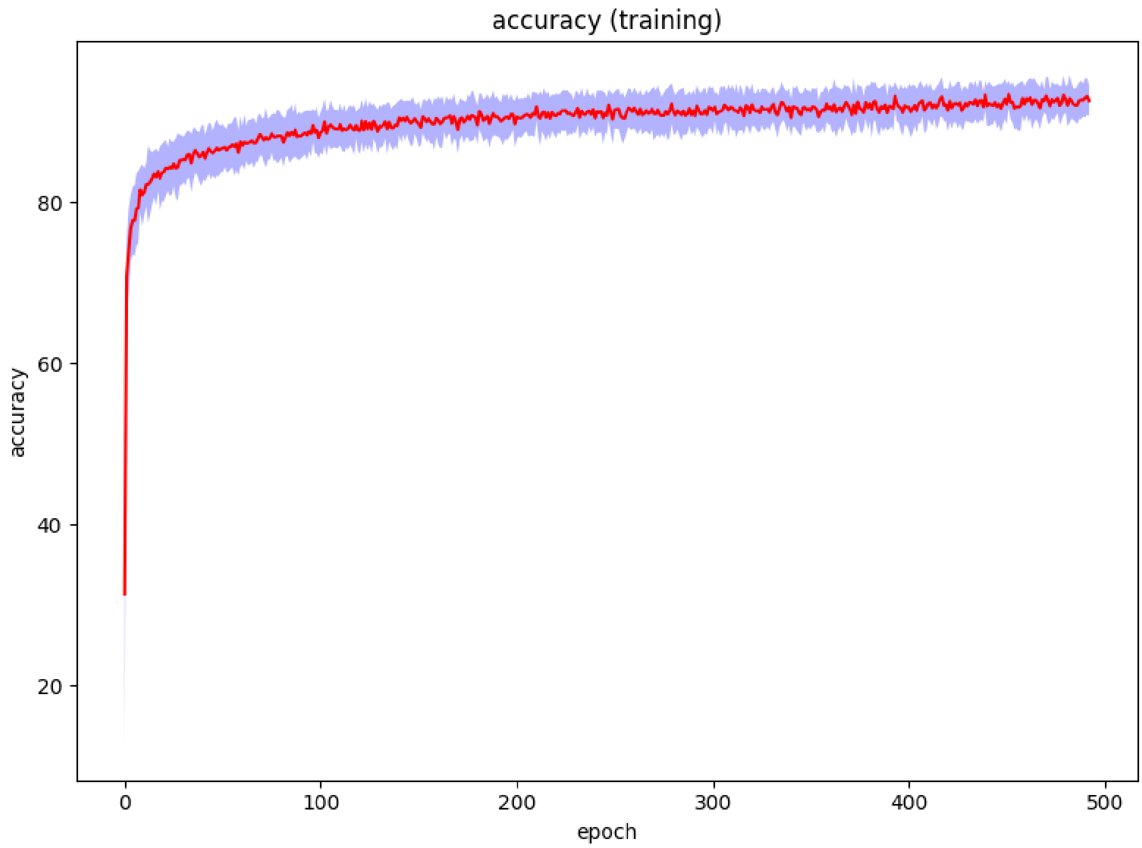

```
########################################################################
#
# RESULT # 05
#
########################################################################
```


loss - data fidelity (testing)
loss
epoch
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500

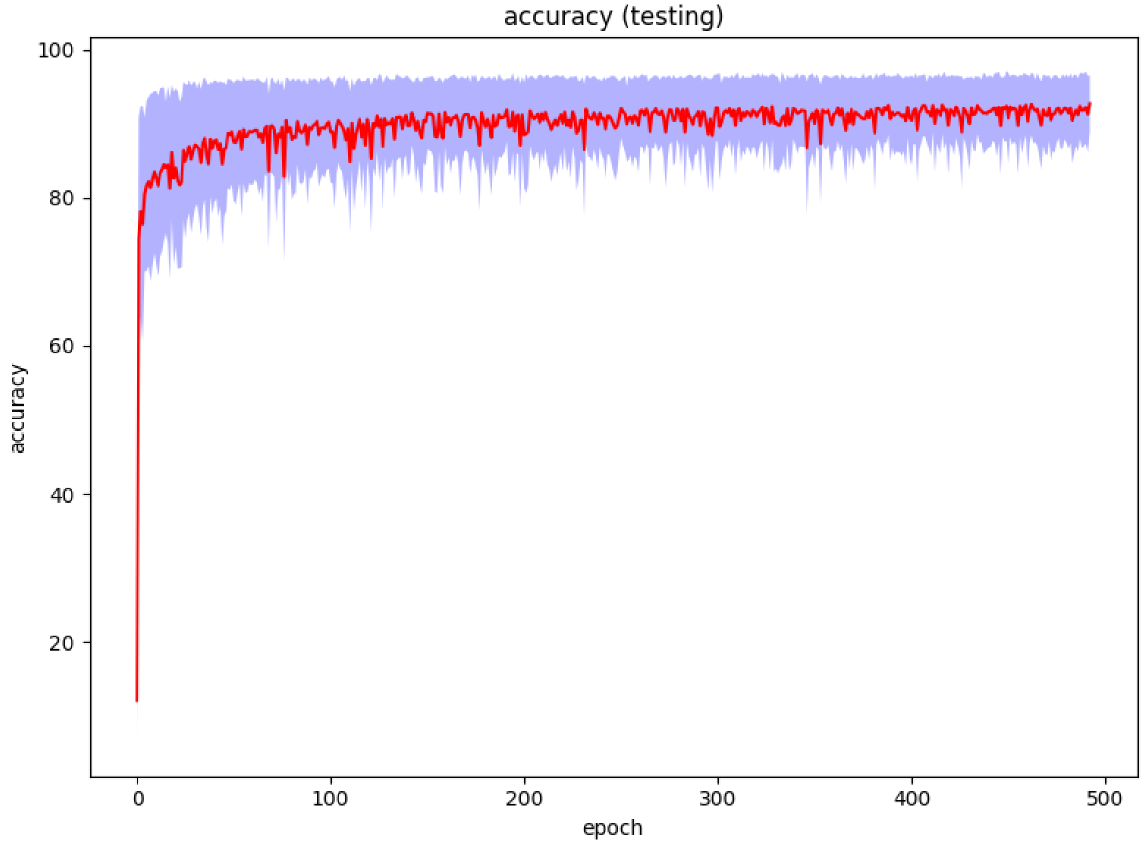
```
####################################################################
############
#
# RESULT # 06
#
####################################################################
############
```


loss - regularization (testing)
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
loss
0
100
200
300
400
500
epoch

```
########################################################################
#
# RESULT # 07
#
########################################################################
```

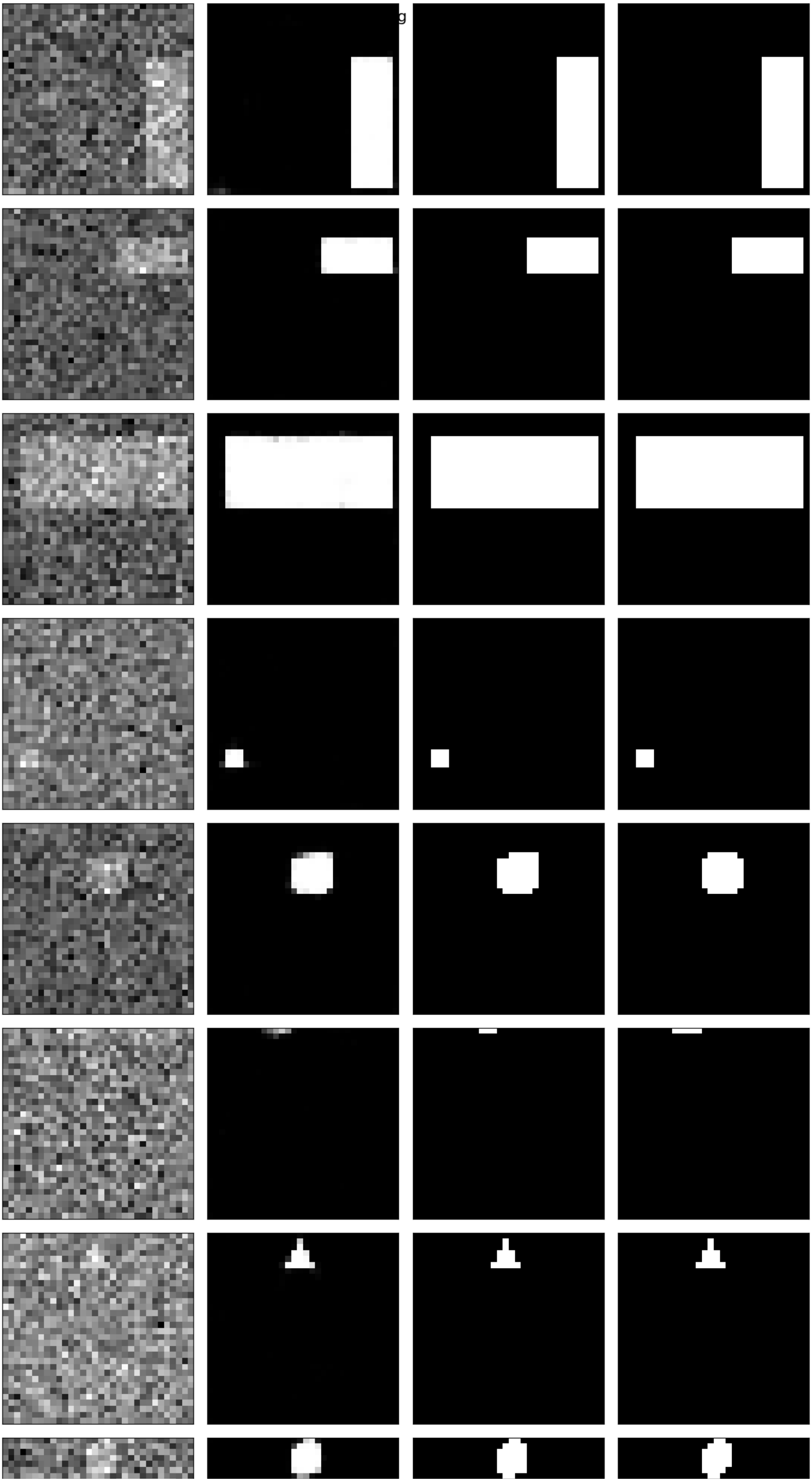

accuracy (training)
accuracy
epoch
80
60
40
20
0
100
200
300
400
500

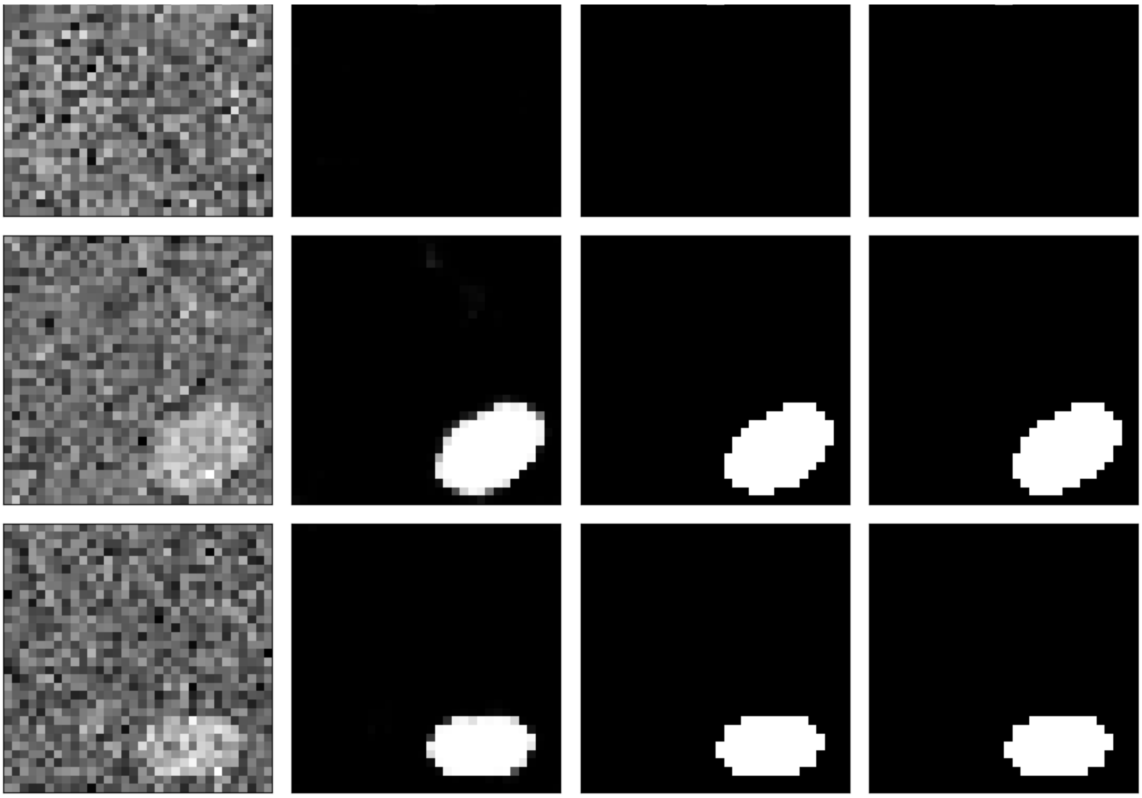
```
########################################################################
#
# RESULT # 08
#
########################################################################
```



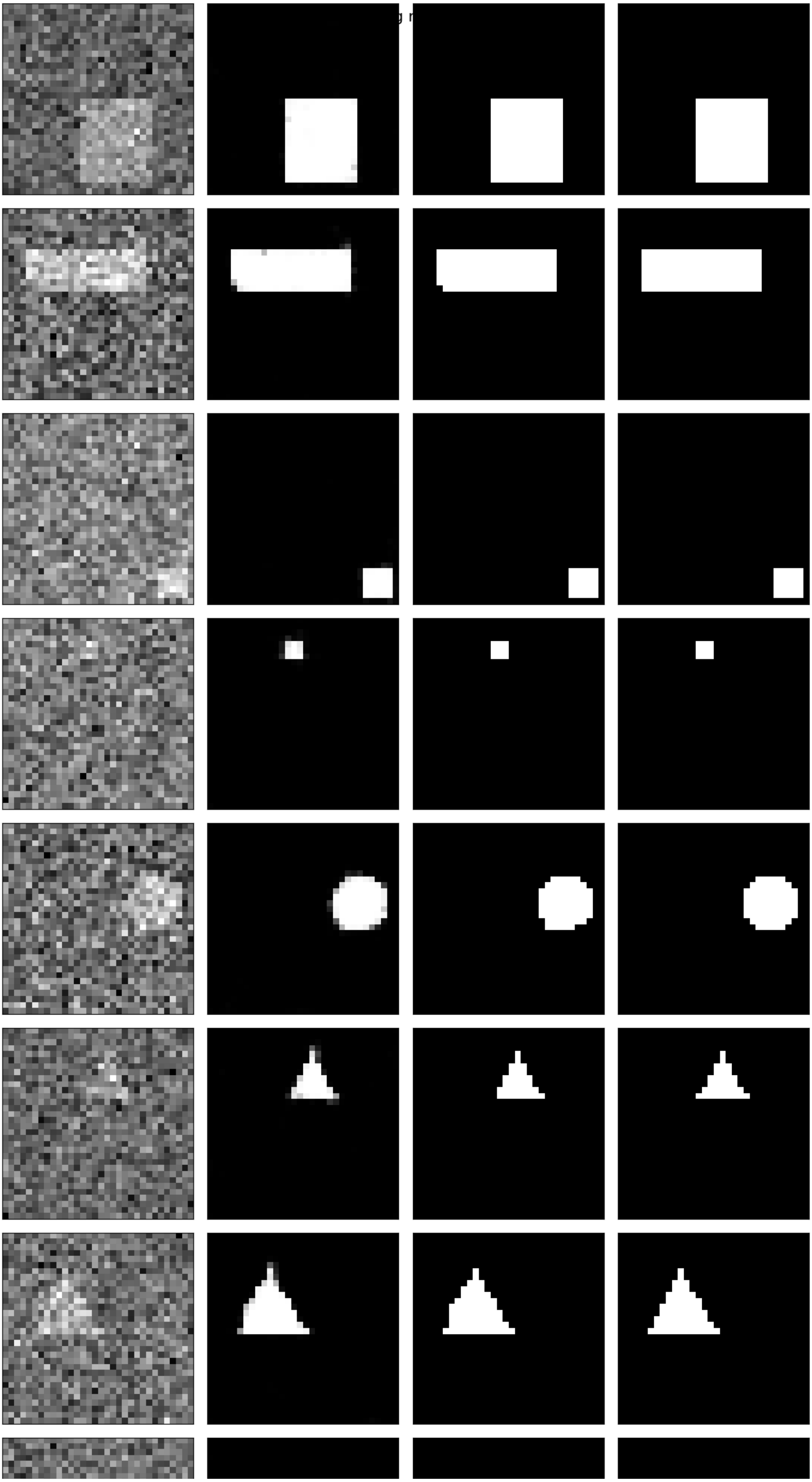
accuracy (testing)
accuracy
epoch
100
80
60
40
20
0
100
200
300
400
500

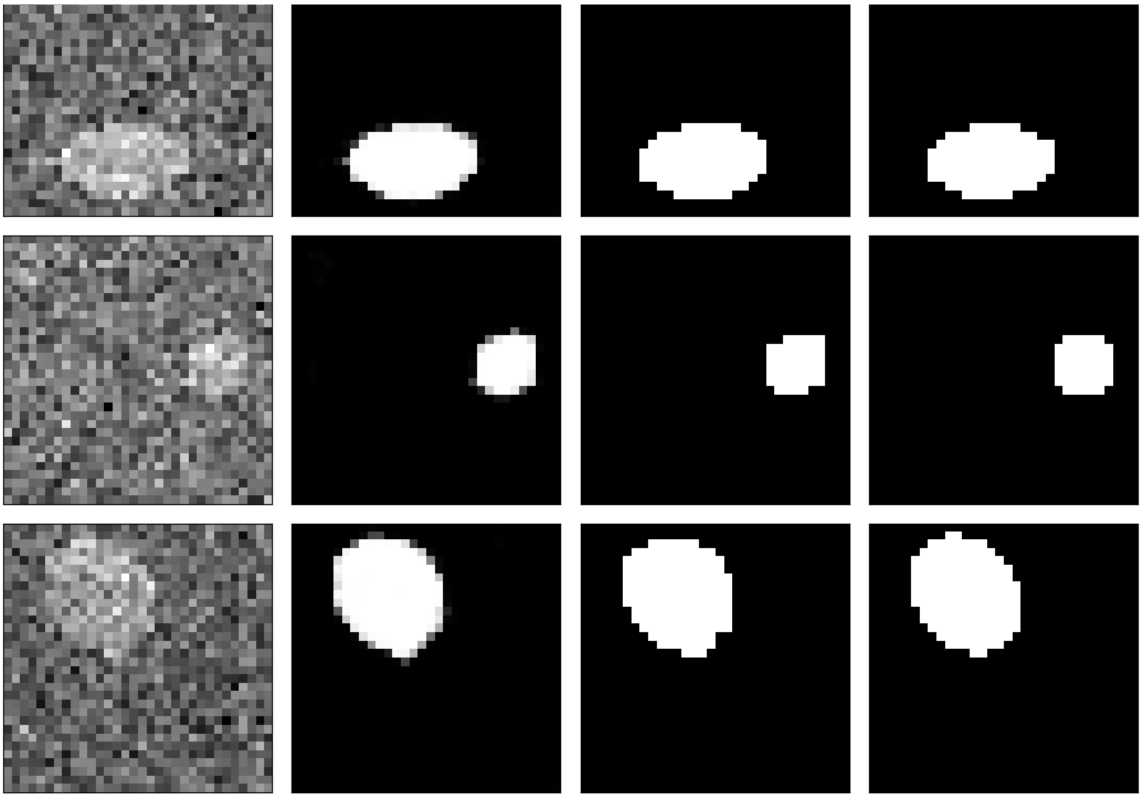

```
########################################################################
############
#
# RESULT # 09
#
########################################################################
############
```





```
##########################################################################
#
# RESULT # 10
#
##########################################################################
```





```
################################################################################
#
# RESULT # 11
#
################################################################################

final training accuracy = 92.61118317

################################################################################
#
# RESULT # 12
#
################################################################################

final testing accuracy = 92.76325989
```

In [ ]: