In [*]:
```
import snntorch as snn
import torch.nn as nn
from torchvision import datasets, transforms
from snntorch import utils, spikegen
from torch.utils.data import DataLoader, TensorDataset, Dataset
from snntorch import surrogate
from snntorch import functional as SF
from snntorch import spikeplot as splt
import matplotlib.pyplot as plt
import torch
import numpy as np
from skorch import NeuralNetClassifier
from sklearn.model_selection import GridSearchCV
```

Data transform and download

In [94]:
```
class AddGaussianNoise(object):
    def __init__(self, mean=0., std=1.):
        self.std = std
        self.mean = mean

    def __call__(self, tensor):
        return tensor + torch.randn(tensor.size()) * self.std + self.mean

    def __repr__(self):
        return self.__class__.__name__ + '(mean={0}, std={1})'.format(self.mean, self.
class AddSaltPepperNoise(object):
    def __init__(self, salt_prob=0.01, pepper_prob=0.01):
        self.salt_prob = salt_prob
        self.pepper_prob = pepper_prob

    def __call__(self, tensor):
        random_tensor = torch.rand(tensor.shape)
        salt_mask = random_tensor < self.salt_prob
        pepper_mask = (random_tensor > (1-self.pepper_prob))
        salt_pepper_noise = torch.zeros_like(tensor)
        salt_pepper_noise = salt_pepper_noise - pepper_mask.float() + salt_mask.float(
        return tensor + salt_pepper_noise

    def __repr__(self):
        return self.__class__.__name__ + '(salt_prob={0}, pepper_prob={1})'.format(sel

transform1 = transforms.Compose([
    transforms.Resize((28, 28)),
    transforms.Grayscale(),
    transforms.ToTensor(),
    transforms.Normalize((0,),(1,)),
    transforms.RandomRotation([-10, 10])
])
transform2 = transforms.Compose([
    transforms.Resize((28, 28)),
    transforms.Grayscale(),
    transforms.ToTensor(),
    transforms.Normalize((0,),(1,)),
    transforms.RandomRotation([-10, 10]),
    AddGaussianNoise(0,0.3)
])
transform3 = transforms.Compose([
    transforms.Resize((28, 28)),
    transforms.Grayscale(),
    transforms.ToTensor(),
    transforms.Normalize((0,),(1,)),
    transforms.RandomRotation([-10, 10]),
    AddSaltPepperNoise(0.1, 0.1)
])

batch_size = 128
train_path = '/final_data/mnist/train'
test_path = '/final_data/mnist/test'
num_classes = 10
mnist_train_ = datasets.MNIST(train_path, train=True, download=True, transform=transf
```

```
mnist_test_ = datasets.MNIST(test_path, train=False, download=True, transform=transfo
```

Take a subset of data to save time

In [19]:
```
subset = 5
mnist_train = utils.data_subset(mnist_train_, subset)
mnist_test = utils.data_subset(mnist_test_, subset)
print(mnist_train)
train_loader = DataLoader(mnist_train, batch_size = batch_size,shuffle=True)
test_loader = DataLoader(mnist_test, batch_size = batch_size,shuffle=True)
print(len(train_loader))
print(len(test_loader))
```

```
Dataset MNIST
    Number of datapoints: 12000
    Root location: /final_data/mnist/train
    Split: Train
    StandardTransform
Transform: Compose(
               Resize(size=(28, 28), interpolation=bilinear, max_size=None, antialia
s=None)
               Grayscale(num_output_channels=1)
               ToTensor()
               Normalize(mean=(0,), std=(1,))
               RandomRotation(degrees=[-10.0, 10.0], interpolation=nearest, expand=F
alse, fill=0)
               AddGaussianNoise(mean=0, std=0.3)
           )
94
16
```

train a non-spiking cnn

In [27]:
```
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
# reset the weight when needed.
def weight_reset(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
        m.reset_parameters()
```

In [70]:
```
num_classes=10
learning_rate = 0.02
weight_decay = 0.05
num_epochs = 7
class ConvNeuralNet(nn.Module):
    def __init__(self,num_classes):
        super(ConvNeuralNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 12, 5)
        self.max_pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(12,32, 5)
        self.max_pool2 = nn.MaxPool2d(2)
        self.ft = nn.Flatten()
        self.relu1 = nn.ReLU()
        self.fc = nn.Linear(32*4*4, num_classes)
    def forward(self,x):
        out = self.conv1(x)
        out = self.max_pool1(out)
        out = self.conv2(out)
        out = self.max_pool2(out)
        out = self.ft(out)
        out = self.relu1(out)
        out = self.fc(out)
        return out
model = ConvNeuralNet(num_classes)
weight_reset(model)
model = model.to(device)
criterion = nn.CrossEntropyLoss()
# optimizer = torch.optim.Adam(model.parameters(), lr=2e-2, betas=(0.9, 0.999))
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=weight_
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(iter(train_loader)):
        images = images.to(device)
        labels = labels.to(device)
#         print(images.size())
        outputs = model(images)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1,num_epochs, loss.item()))
```

```
Epoch [1/7], Loss: 0.4169
Epoch [2/7], Loss: 0.2706
Epoch [3/7], Loss: 0.3613
Epoch [4/7], Loss: 0.2577
Epoch [5/7], Loss: 0.3217
Epoch [6/7], Loss: 0.3600
Epoch [7/7], Loss: 0.2386
```

In [71]:
```
# testing with CNN
with torch.no_grad():
    correct = 0
    total = 0
    for i, (data, label) in enumerate(iter(test_loader)):
        data = data.to(device)
        label = label.to(device)
        output = model(data)
        _, predicted = torch.max(output.data,1)
        total += label.size(0)
        correct += (predicted==label).sum().item()
    print('Accuracy of the network on the {} train images: {}%'.format(12000, 100*corre
```

```
Accuracy of the network on the 12000 train images: 89.95%
```

Rate coding

In [20]:
```
# num_steps =100
# train_event = []
# train_target = []
# test_event = []
# test_target = []
# for i, (data, target) in enumerate(iter(train_loader)):
# #     print(data.size())
#     train_spike_data = spikegen.rate(data,num_steps=num_steps)
#     target = target.unsqueeze(0).expand(num_steps, -1)
#     if i == 0:
#         train_event = train_spike_data
#         train_target = target

#     else:
#         try:
#             train_event = torch.cat((train_event, train_spike_data), dim=0)
#             train_target = torch.cat((train_target, target), dim=0)
#         except:
#             continue
# # train_event = spikegen.rate(train_event,num_steps=num_steps)
# print(train_event.size())
# #     train_event.append(train_spike_data)
# #     train_target.append(target)
# # for i, (data, target) in enumerate(iter(test_loader)):
# #     test_spike_data = spikegen.rate(data,num_steps=num_steps)
# #     target = target.unsqueeze(0).expand(num_steps, -1)
# #     test_event.append(test_spike_data)
# #     test_target.append(target)
# for i, (data, target) in enumerate(iter(test_loader)):
#     test_spike_data = spikegen.rate(data,num_steps=num_steps)
#     target = target.unsqueeze(0).expand(num_steps, -1)
#     if i == 0:
#         test_event = test_spike_data
#         test_target = target

#     else:
#         try:
#             test_event = torch.cat((test_event, test_spike_data), dim=0)
#             test_target = torch.cat((test_target, target), dim=0)
#         except:
#             continue
class CustomTensorDataset(Dataset):
    def __init__(self, data_list, target_list):
        self.data_list = data_list
        self.target_list = target_list

    def __getitem__(self, index):
        return self.data_list[index], self.target_list[index]

    def __len__(self):
        return len(self.data_list)

# Use your lists of tensors to create a Dataset
# def add_noise(inputs,noise_factor=0.3):
#      noisy = inputs+torch.randn_like(inputs) * noise_factor
#      noisy = torch.clip(noisy,0.,1.)
#      return noisy
```

```
num_steps =150
train_event = []
train_target = []
test_event = []
test_target = []
for i, (data, target) in enumerate(iter(train_loader)):
#     data = add_noise(data, 0.3)
    train_spike_data = spikegen.rate(data,num_steps=num_steps)
    target = target.unsqueeze(0).expand(num_steps, -1)
    train_event.append(train_spike_data)
    train_target.append(target)
for i, (data, target) in enumerate(iter(test_loader)):
#     data = add_noise(data, 0.3)
    test_spike_data = spikegen.rate(data,num_steps=num_steps)
    target = target.unsqueeze(0).expand(num_steps, -1)
    test_event.append(test_spike_data)
    test_target.append(target)
# train_data = iter(train_loader)
# clean out the non-consistent data
train_event.pop()
train_target.pop()
# print(len(test_event))
test_event.pop()
# print(len(test_event))
test_target.pop()
# print(test_event)
train_event = torch.stack(train_event)
test_event = torch.stack(test_event)
train_target = torch.stack(train_target)
test_target = torch.stack(test_target)
# # train_event, train_target = next(train_data)
# test_data = iter(test_loader)
# test_event, test_target = next(test_data)
# print('1')
train_data = CustomTensorDataset(train_event, train_target)
test_data = CustomTensorDataset(test_event, test_target)
# train_data = TensorDataset(train_event,train_target)
# test_data = TensorDataset(test_event,test_target)
# manually set the dataset and adjust the batchsize
final_train_loader = DataLoader(train_data, batch_size = 1,shuffle=True)
final_test_loader = DataLoader(test_data, batch_size = 1,shuffle=True)
print(len(final_train_loader))
```

```
93
```

dataset for CNN

In [ ]:

```

```

In [23]:

```
event, target = next(iter(final_train_loader))
print(event.size())
```

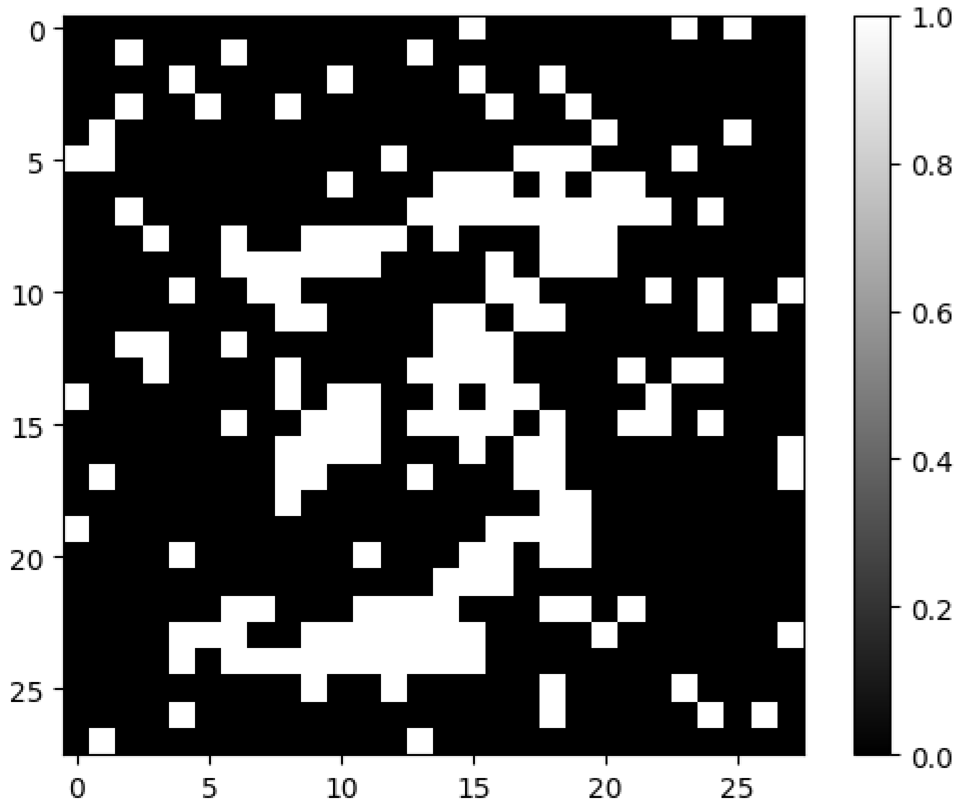
```
torch.Size([1, 150, 128, 1, 28, 28])
```

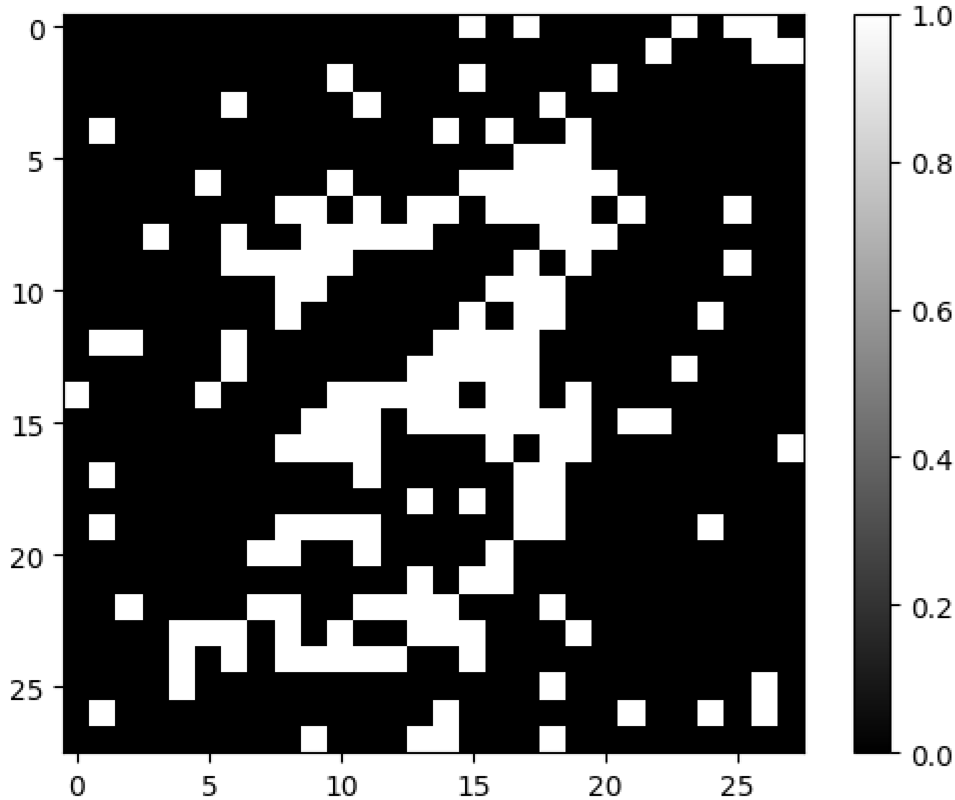
In [24]:
```
# train_data = iter(train_loader)
# clean out the non-consistent data
# train_event.pop()
# train_target.pop()
# print(len(test_event))
# test_event.pop()
# # print(len(test_event))
# test_target.pop()
# print(test_event)
# train_event = torch.stack(train_event)
# test_event = torch.stack(test_event)
# train_target = torch.stack(train_target)
# test_target = torch.stack(test_target)
# # train_event, train_target = next(train_data)
# test_data = iter(test_loader)
# test_event, test_target = next(test_data)
# print(train_event.size())
# train_event = train_event.squeeze(0)
# test_event = test_event.squeeze(0)
# train_data = TensorDataset(train_event,train_target)
# test_data = TensorDataset(test_event,test_target)
# final_train_loader = DataLoader(train_data, batch_size = batch_size,shuffle=True )
# final_test_loader = DataLoader(test_data, batch_size = batch_size,shuffle=True )
# print(train_spike_data.shape)
```

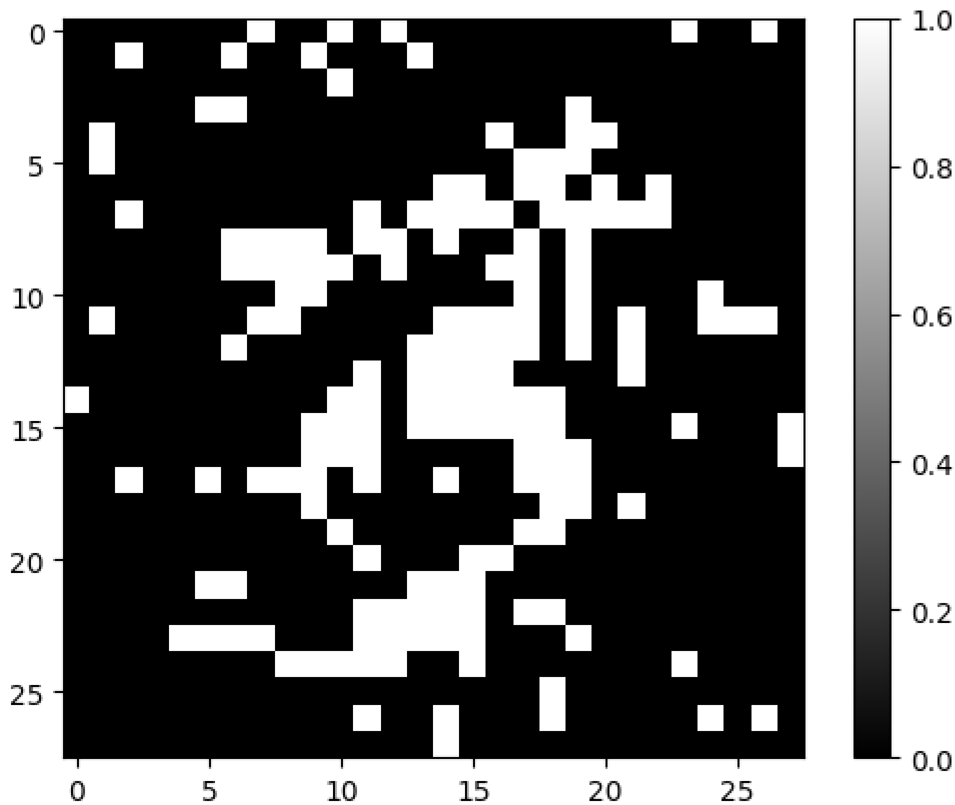
In [25]:
```
print(len(final_train_loader))
```

93

In [26]:
```
for i in range(3):
    spike_image1 = train_spike_data[i][3][0]

# spike_image2 = train_spike_data[2][4][0]
# spike_image3 = train_spike_data[3][4][0]
    plt.imshow(spike_image1, cmap='gray')
# plt.imshow(spike_image2, cmap='gray')
# plt.imshow(spike_image3, cmap='gray')


    plt.colorbar()
    plt.show()
```







build network

In [85]:
```
spike_grad = surrogate.atan()
beta = 0.1

#Initialize Network
net = nn.Sequential(nn.Conv2d(1, 12, 5),
                    snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True, lear
                    nn.MaxPool2d(2),
                    nn.Conv2d(12, 32, 5),
                    snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True, lear
                    nn.MaxPool2d(2),
                    nn.Flatten(),
                    nn.Linear(32*4*4, 10),
                    snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True, outp
                    ).to(device)
# net = nn.Sequential(nn.Conv2d(1, 12, 5),
#                     snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True),
#                     nn.MaxPool2d(2),
#                     nn.Conv2d(12, 32, 5),
#                     snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True),
#                     nn.MaxPool2d(2),
#                     nn.Conv2d(32, 64, 3),  # new convolutional layer
#                     snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True),
#                     nn.MaxPool2d(2),  # new pooling layer
#                     nn.Flatten(),
#                     nn.Linear(64*1*1, 100),  # adjusted for new layer sizes
#                     snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True),
#                     nn.Linear(100, 10),  # new linear layer
#                     snn.Leaky(beta=beta, spike_grad=spike_grad, init_hidden=True, out
#                     ).to(device)
def weight_reset(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
        m.reset_parameters()

net.apply(weight_reset)
```

Out[85]:
```
Sequential(
  (0): Conv2d(1, 12, kernel_size=(5, 5), stride=(1, 1))
  (1): Leaky()
  (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (3): Conv2d(12, 32, kernel_size=(5, 5), stride=(1, 1))
  (4): Leaky()
  (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (6): Flatten(start_dim=1, end_dim=-1)
  (7): Linear(in_features=512, out_features=10, bias=True)
  (8): Leaky()
)
```

In [86]:
```
def forward_pass(net, data):
  spk_rec = []
  utils.reset(net)  # resets hidden states for all LIF neurons in net

  for step in range(data.size(0)):  # data.size(0) = number of time steps
      spk_out, mem_out = net(data[step])
      spk_rec.append(spk_out)

  return torch.stack(spk_rec)
```

In [87]:
```

# optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate_snn, weight_decay=weig

loss_fn = SF.mse_count_loss(correct_rate=0.8, incorrect_rate=0.2)
```

In [88]:
```
def val_accuracy(net, data, label):
    net.eval()
    accur_list = []
    with torch.no_grad():
        for data, labels in zip(data, label):
            data = data
            labels = labels
            data = data.to(device)
            labels = labels.to(device)
            output = forward_pass(net, data)
            accur = SF.accuracy_rate(output, labels[0])
            accur_list.append(accur)


    return sum(accur_list)/len(accur_list)
def test_accuracy(net, dataloader):
    net.eval()
    accur_list = []
    with torch.no_grad():
        for data, labels in iter(dataloader):
            data = data[0]
            labels = labels[0]
            data = data.to(device)
            labels = labels.to(device)
            output = forward_pass(net, data)
            accur = SF.accuracy_rate(output, labels[0])
            accur_list.append(accur)


    return sum(accur_list)/len(accur_list)
```

grid search hyperparameter tuning

In [ ]:
```
```

In [89]:
```
learning_rate = [0.01, 0.02, 0.04, 0.06]
testing_accuracy = []
for i in learning_rate:
    net.apply(weight_reset)
    learning_rate_snn = i
    optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate_snn, betas=(0.9, 0
    num_epochs_snn = 5
    loss_hist = []
    acc_hist = []
    val_acc_list = []
    validation_iter = len(final_train_loader)-int(len(final_train_loader)*0.1)
    print(validation_iter)
    # training loop
    for epoch in range(num_epochs_snn):
        for i, (data, targets) in enumerate(iter(final_train_loader)):
            if i < validation_iter:
                data = data[0]
                targets = targets[0]
                net.train()
                data = data.to(device)
                targets= targets[0] #set the target to be int rather than list
                targets = targets.to(device)
                net.train()
                spk_rec = forward_pass(net, data)
                loss_val = loss_fn(spk_rec, targets)

                # Gradient calculation + weight update
                optimizer.zero_grad()
                loss_val.backward()
                optimizer.step()

                # Store loss history for future plotting
                loss_hist.append(loss_val.item())

                print(f"Epoch {epoch}, Iteration {i} \nTrain Loss: {loss_val.item():.2f

                acc = SF.accuracy_rate(spk_rec, targets)
                acc_hist.append(acc)
            else:
                validation_accuracy = val_accuracy(net, data, targets)
                val_acc_list.append(validation_accuracy)
                print(f"validation accuracy: {validation_accuracy*100:.2f}%\n")
            print(f"Accuracy: {acc * 100:.2f}%\n")
    test_accuracy = test_accuracy(net, final_test_loader)
    testing_accuracy.append(test_accuracy)
```

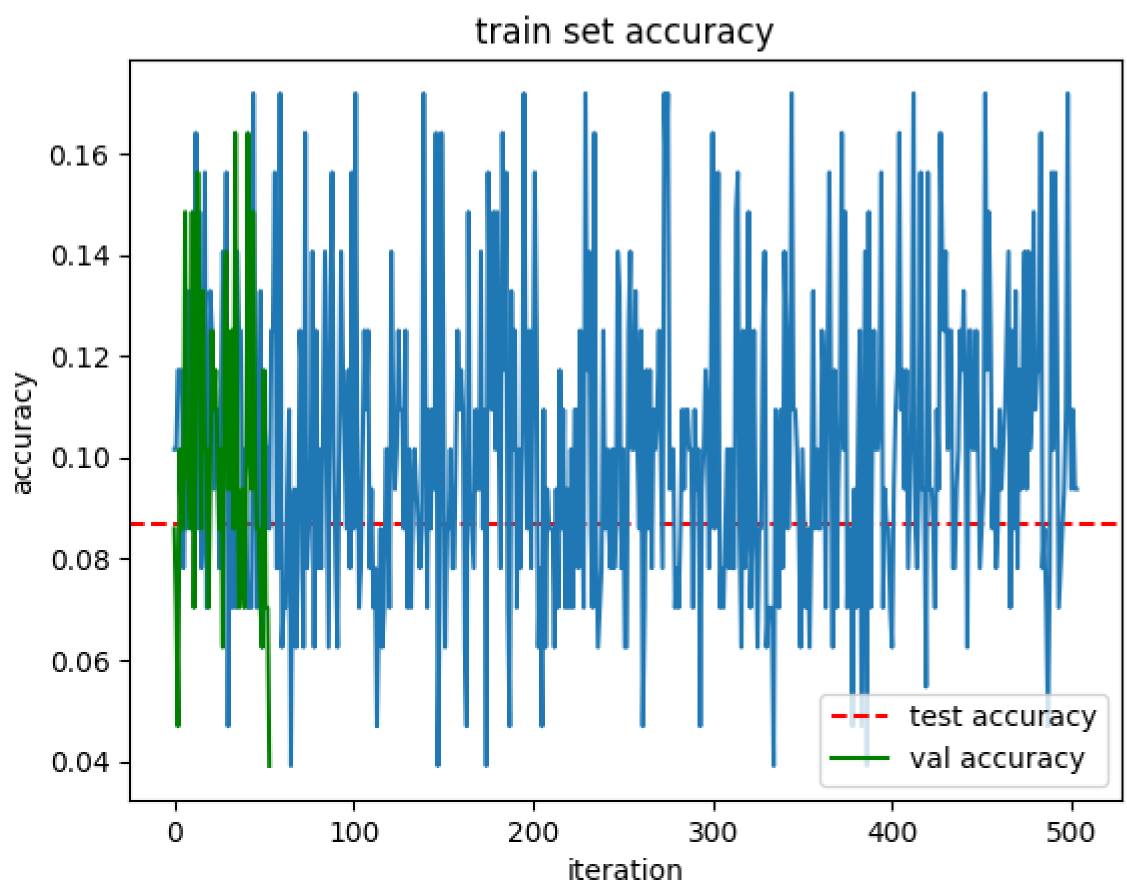
```
Accuracy: 9.38%

validation accuracy: 7.03%

Accuracy: 9.38%

validation accuracy: 6.25%

Accuracy: 9.38%

validation accuracy: 11.72%

Accuracy: 9.38%

validation accuracy: 7.03%

Accuracy: 9.38%

validation accuracy: 7.03%
```

testing set

In [90]:
```
test_accuracy = test_accuracy(net, final_test_loader)
print(test_accuracy)
fig = plt.figure(facecolor='w')
plt.axhline(y=test_accuracy, color='r', linestyle='--',label='test accuracy')
plt.plot(acc_hist)
plt.plot(val_acc_list, color='g', label='val accuracy')
# plt.plot(loss_hist, color='b', label='train loss')
plt.title('train set accuracy')
plt.xlabel("iteration")
plt.ylabel('accuracy')
plt.legend()
plt.show
```

```
0.08697916666666666
```

Out[90]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



In [ ]:

In [ ]: