COMP 4211 PA2 REPORT

CHEN, Yifei 20328874

Part 0: Preparation Function for Implementing the tasks

The libraries I imported in this assignment:

```
from torchvision.datasets import EMNIST
from torch.utils.data import ConcatDataset, Subset
from torchvision.transforms import ToTensor, Compose
import numpy as np

# the library imported by myself
import torch
import torch.nn as nn

import os
from tensorboardX import SummaryWriter
from torch.utils.data import DataLoader
from torch.optim import Adam, SGD # just choose which to use
from torchvision.utils import make_grid
import matplotlib.pyplot as plt
```

Helper get_dataset functions (From tuto):

```
def get_datasets(split='balanced', save=False):
  download_folder = './data'

  transform = Compose([ToTensor()])

  dataset = ConcatDataset([EMNIST(root=download_folder, split=split, download=True, train=False, transform=transform),
                           EMNIST(root=download_folder, split=split, download=True, train=True, transform=transform)])

  # save already = false, the program will not go into the below part, I just leave it.
  if save:
    random_seed = 4211 # do not change
    n_samples = len(dataset)
    eval_size = 0.2
    indices = list(range(n_samples))
    split = int(np.floor(eval_size * n_samples))

    np.random.seed(random_seed)
    np.random.shuffle(indices)

    train_indices, eval_indices = indices[split:], indices[:split]

    # cut to half
    train_indices = train_indices[:len(train_indices)//2]
    eval_indices = eval_indices[:len(eval_indices)//2]

    np.savez('train_test_split.npz', train=train_indices, test=eval_indices)

  # This is what really happen
  # load train test split indices
  else:
    with np.load('./train_test_split.npz') as f:
      train_indices = f['train']
      eval_indices = f['test']

  train_dataset = Subset(dataset, indices=train_indices)
  eval_dataset = Subset(dataset, indices=eval_indices)

  return train_dataset, eval_dataset
```

The assignment was implemented in Python (import torch), aided by JupyterLab and Tensorboard.

Part 1: CNN Classifiers (Learning from scratch)

1.The first thing I did is to construct the CNN class as required, which was shown below:

```python
class Cnn_Scratch_Classifier(nn.Module):
  def __init__(self, n_hidden):
    super(Cnn_Scratch_Classifier, self).__init__()

    # in_data size: (batch_size, 1, 28, 28)
    self.cnn_layers = nn.Sequential(
      # conv1_out size: (batch_size, 4, 26, 26)
      nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, stride=1, padding=0),
      nn.ReLU(),
      # conv2_out size: (batch_size, 8, 12, 12)
      nn.Conv2d(in_channels=4, out_channels=8, kernel_size=3, stride=2, padding=0),
      nn.ReLU(),
      # conv3_out size: (batch_size, 16, 5, 5)
      nn.Conv2d(in_channels=8, out_channels=16, kernel_size=3, stride=2, padding=0),
      nn.ReLU(),
      # MaxPool2d_out size: (batch_size, 16, 3, 3)
      nn.MaxPool2d(kernel_size=3, stride=1, padding=0),
      # conv4_out size: (batch_size, 32, 1, 1)
      nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=0),
      # As required, just after the last convolution layer, logistic function should be used
      nn.Sigmoid()
    )

    # linear layers transforms flattened image features into logits before the softmax layer
    self.linear = nn.Sequential(
      nn.Linear(32, n_hidden),
      nn.ReLU(),
      nn.Linear(n_hidden, 47) # there are 47 predicted_labels
    )

    self.softmax = nn.Softmax(dim=1)
    self.loss_fn = nn.CrossEntropyLoss(reduction='sum') # will be divided by batch size

  def forward(self, in_data):
    img_features = self.cnn_layers(in_data).view(in_data.size(0), 32) # in_data.size(0) == batch_size
    logits = self.linear(img_features)
    return logits

  def loss(self, logits, labels):
    preds = self.softmax(logits) # size (batch_size, 10)
    return self.loss_fn(preds, labels) / logits.size(0) # divided by batch_size


  def top1_accuracy(self, logits, labels):
    # get argmax of logits along dim=1 (this is equivalent to argmax of predicted probabilites)
    predicted_labels = torch.argmax(logits, dim=1, keepdim=False)
    n_corrects = float(predicted_labels.eq(labels).sum(0)) # sum up all the correct predictions
    return int(n_corrects / logits.size(0) * 100) # in percentage

  def top3_accuracy(self, logits, labels):
    n_corrects = 0
    # copy logits for implementation without chage original value
    logits_copy = logits.clone().detach()
    # get argmax of logits along dim=1 (this is equivalent to argmax of predicted probabilites)
    max1 = torch.argmax(logits_copy, dim=1)
    n_corrects = n_corrects + max1.eq(labels).sum(0)
    for i in range(max1.shape[0]):
        logits_copy[i][max1[i]] = -99999

    max2 = torch.argmax(logits_copy, dim=1)
    n_corrects = n_corrects + max2.eq(labels).sum(0)
    for i in range(max2.shape[0]):
        logits_copy[i][max2[i]] = -99999

    max3 = torch.argmax(logits_copy, dim=1)
    n_corrects = float(n_corrects + max3.eq(labels).sum(0))
    for i in range(max3.shape[0]):
        logits_copy[i][max3[i]] = -99999

    return int((n_corrects / logits.size(0)) * 100) # in percentage
```

And I also need a train function which can be called in the later parts (not inside the class definition):

```python
def train(model, loaders, optimizer, writer, n_epochs, ckpt_path, device='cpu'):
  def run_epoch(train_or_eval):
    epoch_loss = 0.
    epoch_acc_1 = 0.
    epoch_acc_3 = 0.

    for i, batch in enumerate(loaders[train_or_eval], 1):
      in_data, labels = batch
      in_data, labels = in_data.to(device), labels.to(device)

      if train_or_eval == 'train':
        optimizer.zero_grad()

      logits = model(in_data)

      batch_loss = model.loss(logits, labels)
      batch_acc_1 = model.top1_accuracy(logits, labels)
      batch_acc_3 = model.top3_accuracy(logits, labels)

      epoch_loss += batch_loss
      epoch_acc_1 += batch_acc_1
      epoch_acc_3 += batch_acc_3

      if train_or_eval == 'train':
        batch_loss.backward()
        optimizer.step()

    epoch_loss /= i
    epoch_acc_1 /= i
    epoch_acc_3 /= i
    print('top1 acc: %s     top3 acc: %s' %(epoch_acc_1,epoch_acc_3))
    print()

    losses[train_or_eval] = epoch_loss
    accs_1[train_or_eval] = epoch_acc_1
    accs_3[train_or_eval] = epoch_acc_3

    if writer is None:
      print('epoch %d %s loss %.4f acc %.4f' % (epoch, train_or_eval, epoch_loss, epoch_acc))
    elif train_or_eval == 'eval':
      writer.add_scalars('%s_loss' % model.__class__.__name__,
                         tag_scalar_dict={'train': losses['train'],
                                          'eval': losses['eval']},
                         global_step=epoch)

      writer.add_scalars('%s_top1_accuracy' % model.__class__.__name__,
                         tag_scalar_dict={'train': accs_1['train'],
                                          'eval': accs_1['eval']},
                         global_step=epoch)

      writer.add_scalars('%s_top3_accuracy' % model.__class__.__name__,  # CnnClassifier or F
                         tag_scalar_dict={'train': accs_3['train'],
                                          'eval': accs_3['eval']},
                         global_step=epoch)

      # For instructional purpose, add images here, just the last in_data
      #if epoch % 10 == 0:
      #  if len(in_data.size()) == 2: # when it is flattened, reshape it
      #    in_data = in_data.view(-1, 1, 28, 28)

      #  img_grid = make_grid(in_data.to('cpu'))
      #  writer.add_image('%s/eval_input' % model.__class__.__name__, img_grid, epoch)

  # main statements
  losses = dict()
  accs_1 = dict()
  accs_3 = dict()

  for epoch in range(1, n_epochs+1):
    print('Epoch %s :' %epoch)
    print('Train:')
    run_epoch('train')
    print('Eval:')
    run_epoch('eval')

    # For instructional purpose, show how to save checkpoints
    if ckpt_path is not None:
      torch.save({
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'epoch': epoch,
        'losses': losses,
        'accs_1': accs_1,
        'accs_3': accs_3
      }, '%s/%d.pt' % (ckpt_path, epoch))
```

2. With the above functions, I could write the main function for CNN Classifier via Scratch Learning: (set the number of epoch = 10, batch size = 32)

```
def cnn_scratch_main(n_hidden,optim,learning_rate):

  gpu = -1 # default
  #lr = args.lr
  lr = learning_rate
  #batch_size = args.batch
  batch_size = 32 # default
  #ckpt_path = args.ckpt
  ckpt_path = './ckpt/cnn'
  #n_epochs = args.epoch
  n_epochs = 10 # default
  #opt_str = args.optim
  opt_str = optim

  ckpt_path = '%s/%s' % (ckpt_path, opt_str)

  if ckpt_path is not None:
    if not(os.path.exists(ckpt_path)):
      os.makedirs(ckpt_path)

  if gpu == -1:
    DEVICE = 'cpu'
  elif torch.cuda.is_available():
    DEVICE = gpu

  #model = CnnClassifier(n_hidden).to(DEVICE)
  # For the task 3.2.1, i need the Cnn_Scratch_Classifier model
  model = Cnn_Scratch_Classifier(n_hidden)

  train_dataset, test_dataset = get_datasets()

  # Here I need to split the train_dataset in to train and eval with the ratio 4:1. I determine to set the last 20% as validatic

  n_samples = len(train_dataset)
  eval_start = 0.8
  split = int(np.floor(eval_start * n_samples))
  indices = list(range(n_samples))
  train_indices, validation_indices = indices[:split], indices[split:]

  dataloaders = {
    'train':  DataLoader(Subset(train_dataset, indices=train_indices),batch_size=batch_size, drop_last=False, shuffle=True),
    'eval':   DataLoader(Subset(train_dataset, indices=validation_indices),batch_size=batch_size, drop_last=False, shuffle=True)
  }

  if opt_str == 'adam':
    opt_class = Adam
  elif opt_str == 'sgd':
    opt_class = SGD
```

```
optimizer = opt_class(model.parameters(), lr=lr)
writer = SummaryWriter('./logs/cnn_scratch/%s_%s_%s' % (opt_str,n_hidden,lr))

train(model, dataloaders, optimizer, writer, n_epochs, ckpt_path, DEVICE)
```

One issue to be noticed here is that I split the training dataset before I started the training with a ratio of 4:1 = Train: Validation. The training dataset and the evaluation dataset would not change at each epoch.

3. Then I implemented the model with the required parameter sets:

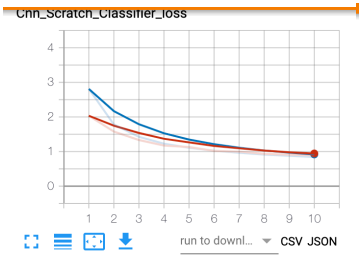
Hyperparaemter	Candidates
H	32, 64
(Optimizer, Learning Rate)	(ADAM, 0.001), (SGD, 0.1), (SGD, 0.01)

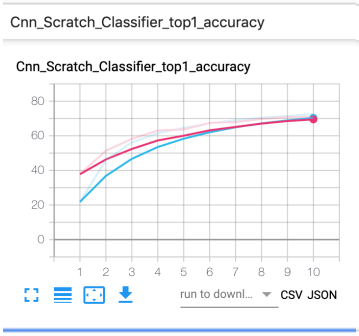
Using the tensor board' s command, I can see the graph at localhost:6006

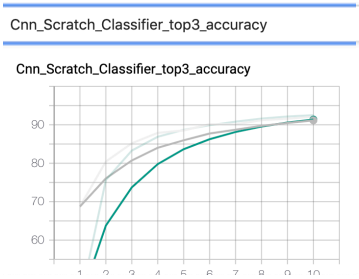
```
(base) YifeideMacBook-Pro:logs chenyifei$ tensorboard --logdir cnn_scratch/
```

The results are shown below:

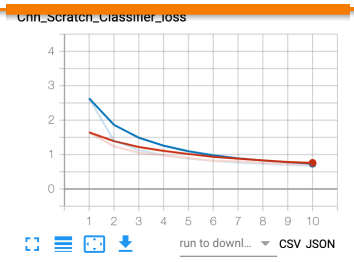
32_adam_0.001

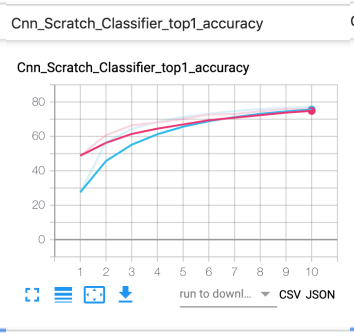


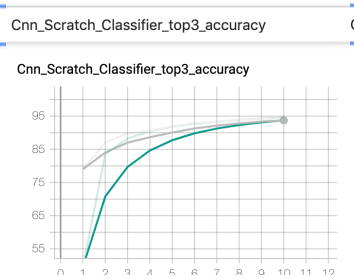




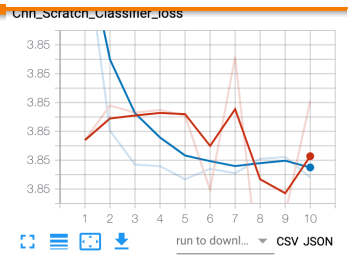
64_adam_0.001

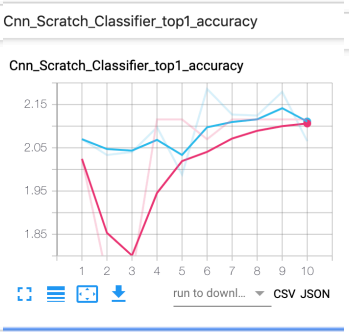


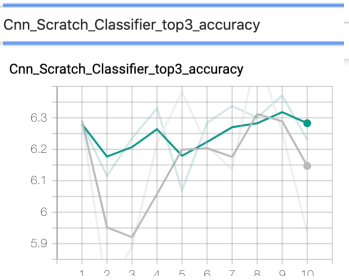




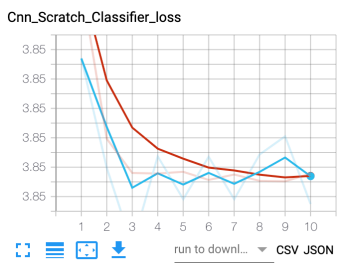
32_sgd_0.1.

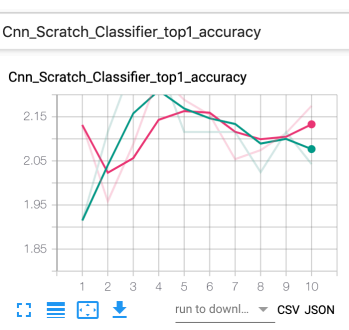


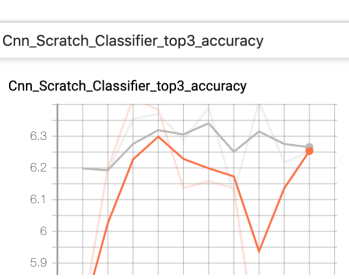




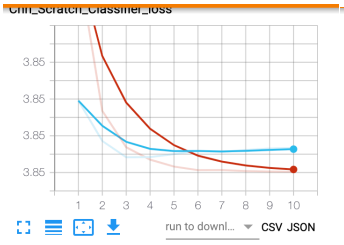
64_sgd_0.1

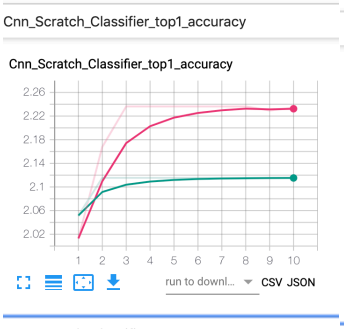


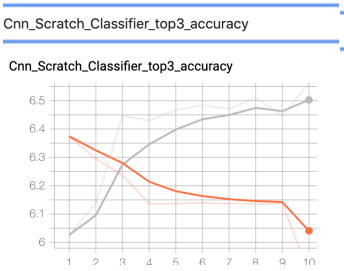




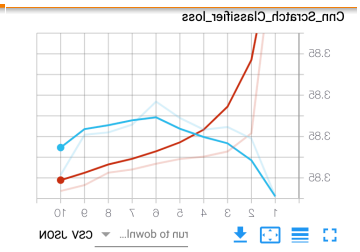
32_sgd_0.01

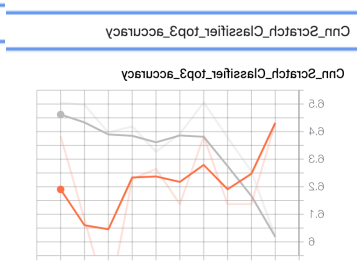






64_sgd_0.01



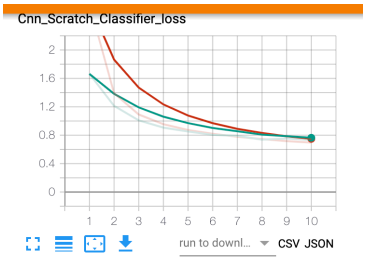


From the data I got, the set{adam,64,0.01} performed best in top1 acc, top3 acc, and loss. Then I chose it as the optimal set for the Testing Phase.

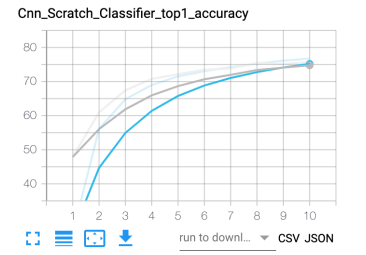
Name	Smoothed	Value	Step	Time	Relative
adam_32_0.001/Cnn_Scratch_Classifier_loss/eval	0.9445	0.8911	10	Tue Apr 9, 20:10:57	2m 0s
adam_32_0.001/Cnn_Scratch_Classifier_loss/train	0.9173	0.8418	10	Tue Apr 9, 20:10:57	2m 0s
adam_64_0.001/Cnn_Scratch_Classifier_loss/eval	0.7574	0.7142	10	Tue Apr 9, 20:21:29	2m 30s
adam_64_0.001/Cnn_Scratch_Classifier_loss/train	0.7355	0.6748	10	Tue Apr 9, 20:21:29	2m 30s
sgd_32_0.01/Cnn_Scratch_Classifier_loss/eval	3.851	3.851	10	Tue Apr 9, 20:39:10	2m 5s
sgd_32_0.01/Cnn_Scratch_Classifier_loss/train	3.85	3.85	10	Tue Apr 9, 20:39:10	2m 5s
sgd_32_0.1/Cnn_Scratch_Classifier_loss/eval	3.851	3.851	10	Tue Apr 9, 20:24:26	2m 1s
sgd_32_0.1/Cnn_Scratch_Classifier_loss/train	3.851	3.851	10	Tue Apr 9, 20:24:26	2m 1s
sgd_64_0.01/Cnn_Scratch_Classifier_loss/eval	3.85	3.85	10	Tue Apr 9, 20:42:47	2m 41s
sgd_64_0.01/Cnn_Scratch_Classifier_loss/train	3.85	3.85	10	Tue Apr 9, 20:42:47	2m 41s
sgd_64_0.1/Cnn_Scratch_Classifier_loss/eval	3.851	3.851	10	Tue Apr 9, 20:31:52	2m 24s
sgd_64_0.1/Cnn_Scratch_Classifier_loss/train	3.851	3.851	10	Tue Apr 9, 20:31:52	2m 24s

4. Test Phasing

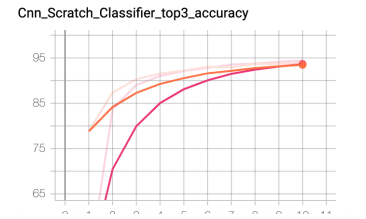
Using the parameter set I chose for this part: train the whole training data set and test for testing set. one of five times' graph is shown below:



Cnn_Scratch_Classifier_top1_accuracy



Cnn_Scratch_Classifier_top3_accuracy



From the five times' test, I got the results:

adam_64_0.001	1st	2nd	3rd	4th	5th	average	std
Top1 acc	75.7913	78.2767	78.4466	76.6578	74.2282	76.68012	1.765781
Top3 acc	93.9757	94.9684	95.0485	94.5461	92.7354	94.25482	0.95006756
loss	0.7316	0.657	0.6503	0.6875	0.8078	0.70684	0.06492459

Part 2: CNN Classifiers (Loading pretrained encoder)

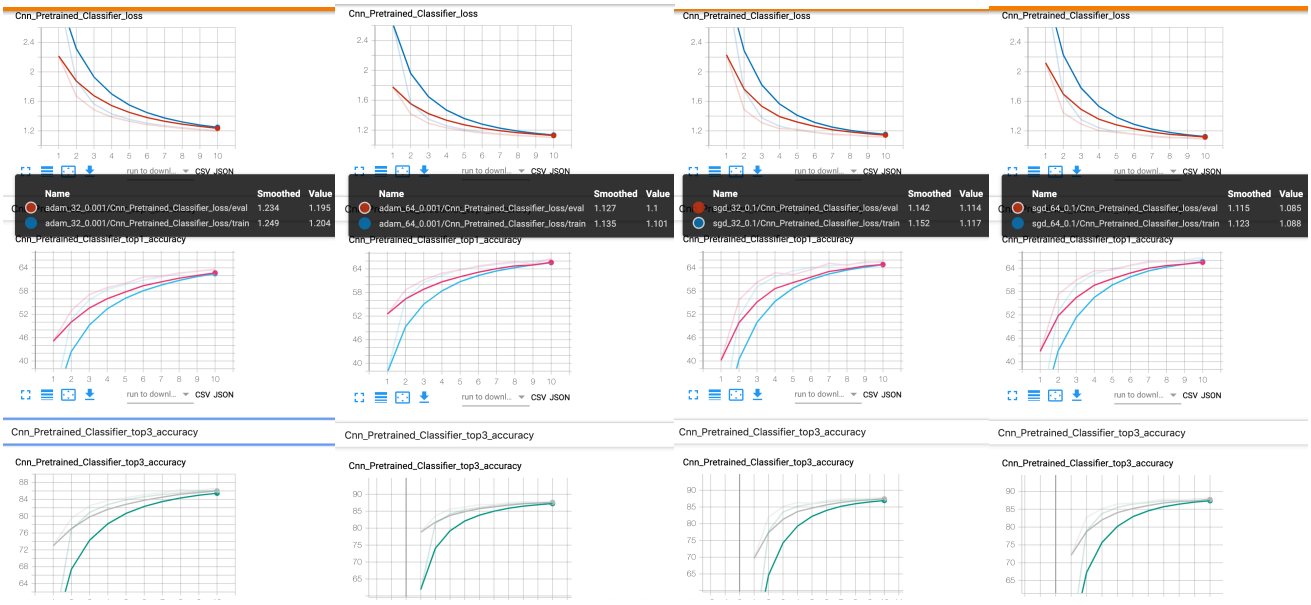
1. For this part, the CNN classifier is different (the encoder is loaded from assigned pt. file)

```
class Cnn_Pretrained_Classifier(nn.Module):
 def __init__(self, n_hidden):

   super(Cnn_Pretrained_Classifier, self).__init__()

   self.cnn_layers = torch.load('pretrained_encoder.pt', map_location = 'cpu')['model']

   # linear layers transforms flattened image features into logits before the softmax layer
   self.linear = nn.Sequential(
     nn.Linear(32, n_hidden),
     nn.ReLU(),
     nn.Linear(n_hidden, 47) # there are 47 predicted_labels
   )

   self.softmax = nn.Softmax(dim=1)
   self.loss_fn = nn.CrossEntropyLoss(reduction='sum') # will be divided by batch size

 def forward(self, in_data):
    img_features = self.cnn_layers(in_data).view(in_data.size(0), 32) # in_data.size(0) == batch_size
    logits = self.linear(img_features)
    return logits

 def loss(self, logits, labels):
    #preds = self.softmax(logits) # size (batch_size, 10)
    return self.loss_fn(logits, labels) / logits.size(0) # divided by batch_size


 def top1_accuracy(self, logits, labels):
   # get argmax of logits along dim=1 (this is equivalent to argmax of predicted probabilites)
   predicted_labels = torch.argmax(logits, dim=1, keepdim=False) # size (batch_size,)
   n_corrects = float(predicted_labels.eq(labels).sum(0)) # sum up all the correct predictions
   return int(n_corrects / logits.size(0) * 100) # in percentage
```

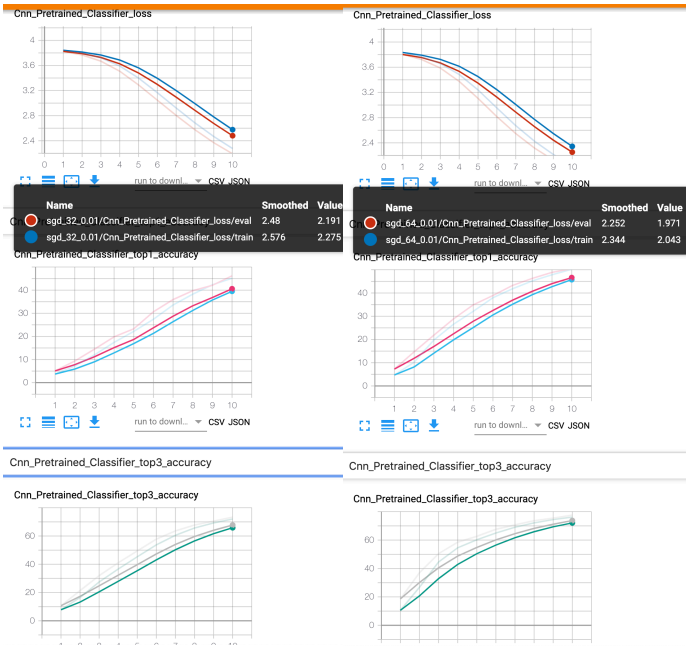
```
def top3_accuracy(self, logits, labels):
    n_corrects = 0
    # copy logits for implementation without chage original value
    logits_copy = logits.clone().detach()
    # get argmax of logits along dim=1 (this is equivalent to argmax of predicted probabilites)
    max1 = torch.argmax(logits_copy, dim=1)
    n_corrects = n_corrects + max1.eq(labels).sum(0)
    for i in range(max1.shape[0]):
        logits_copy[i][max1[i]] = -99999

    max2 = torch.argmax(logits_copy, dim=1)
    n_corrects = n_corrects + max2.eq(labels).sum(0)
    for i in range(max2.shape[0]):
        logits_copy[i][max2[i]] = -99999

    max3 = torch.argmax(logits_copy, dim=1)
    n_corrects = float(n_corrects + max3.eq(labels).sum(0))
    for i in range(max3.shape[0]):
        logits_copy[i][max3[i]] = -99999

    return int(n_corrects / logits.size(0) * 100 )# in percentage
```

2. The procedure is pretty similar with part 1, here I will list the results:

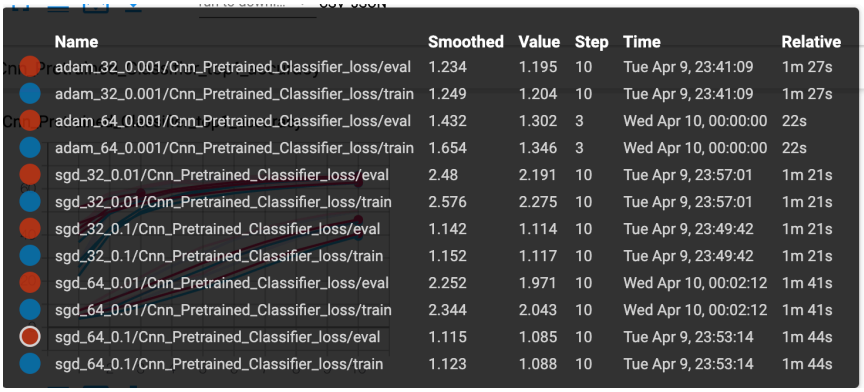
Adam_32_0.001 adam_64_0.001 sgd_32_0.1 sgd_64_0.1



Sgd_32_0.01 sgd_64_0.01



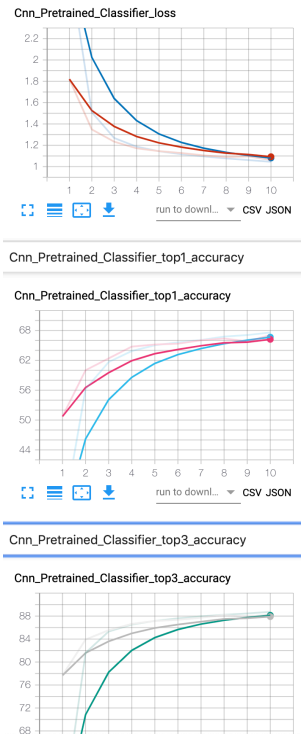
From the result I got, I will choose {sgd_64_0.1}as the optimal set



Name	Smoothed	Value	Step	Time	Relative
adam_32_0.001/Cnn_Pretrained_Classifier_loss/eval	1.234	1.195	10	Tue Apr 9, 23:41:09	1m 27s
adam_32_0.001/Cnn_Pretrained_Classifier_loss/train	1.249	1.204	10	Tue Apr 9, 23:41:09	1m 27s
adam_64_0.001/Cnn_Pretrained_Classifier_loss/eval	1.432	1.302	3	Wed Apr 10, 00:00:00	22s
adam_64_0.001/Cnn_Pretrained_Classifier_loss/train	1.654	1.346	3	Wed Apr 10, 00:00:00	22s
sgd_32_0.01/Cnn_Pretrained_Classifier_loss/eval	2.48	2.191	10	Tue Apr 9, 23:57:01	1m 21s
sgd_32_0.01/Cnn_Pretrained_Classifier_loss/train	2.576	2.275	10	Tue Apr 9, 23:57:01	1m 21s
sgd_32_0.1/Cnn_Pretrained_Classifier_loss/eval	1.142	1.114	10	Tue Apr 9, 23:49:42	1m 21s
sgd_32_0.1/Cnn_Pretrained_Classifier_loss/train	1.152	1.117	10	Tue Apr 9, 23:49:42	1m 21s
sgd_64_0.01/Cnn_Pretrained_Classifier_loss/eval	2.252	1.971	10	Wed Apr 10, 00:02:12	1m 41s
sgd_64_0.01/Cnn_Pretrained_Classifier_loss/train	2.344	2.043	10	Wed Apr 10, 00:02:12	1m 41s
sgd_64_0.1/Cnn_Pretrained_Classifier_loss/eval	1.115	1.085	10	Tue Apr 9, 23:53:14	1m 44s
sgd_64_0.1/Cnn_Pretrained_Classifier_loss/train	1.123	1.088	10	Tue Apr 9, 23:53:14	1m 44s

3. Testing phase

One of the five times' results:



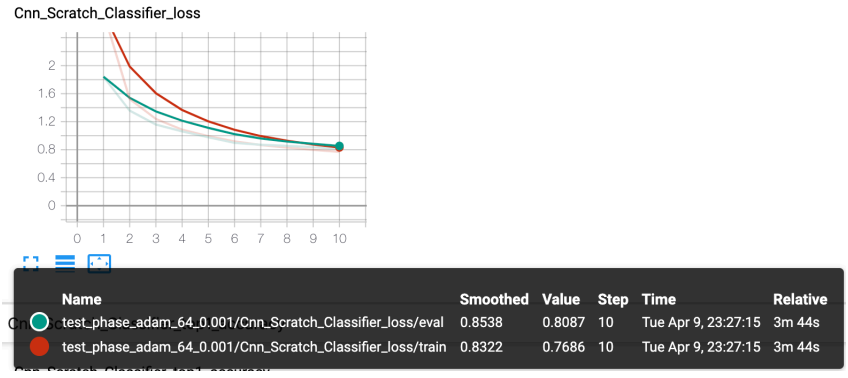
The final results with the best parameter set:

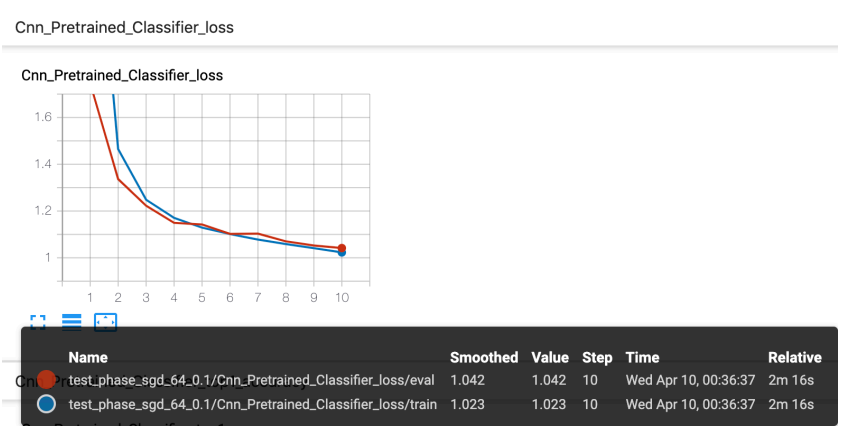
sgd_64_0.1	1st	2nd	3rd	4th	5th	average	std
Top1 acc	67.1019	68.466	67.8617	67.6602	67.9199	67.80194	0.49208935
Top3 acc	88.3495	89.1311	88.7961	88.3447	88.7621	88.6767	0.33365878
loss	1.0621	1.0202	1.0344	1.062	1.0417	1.04408	0.01813469

Part3: Analysis of Scratch vs. Pretrained

1. Training Time:

   From the results shown on tensorboard, I find that the training time using scratch is longer than using the pretrained model. For instance, both running 10 epochs, the training time used by way of scratch is 3m44s in the figure, while the training time used by way of pretrained model is 2m16s. I think the difference comes from the optimizer' s computation on the encoder: In the model of scratch, the weights of the encoder needs to be updated every epoch, which need more time for computation. However, when using the pretrained model, the weights in the encoder is fixed (grad_requires = false), which will save time for training.

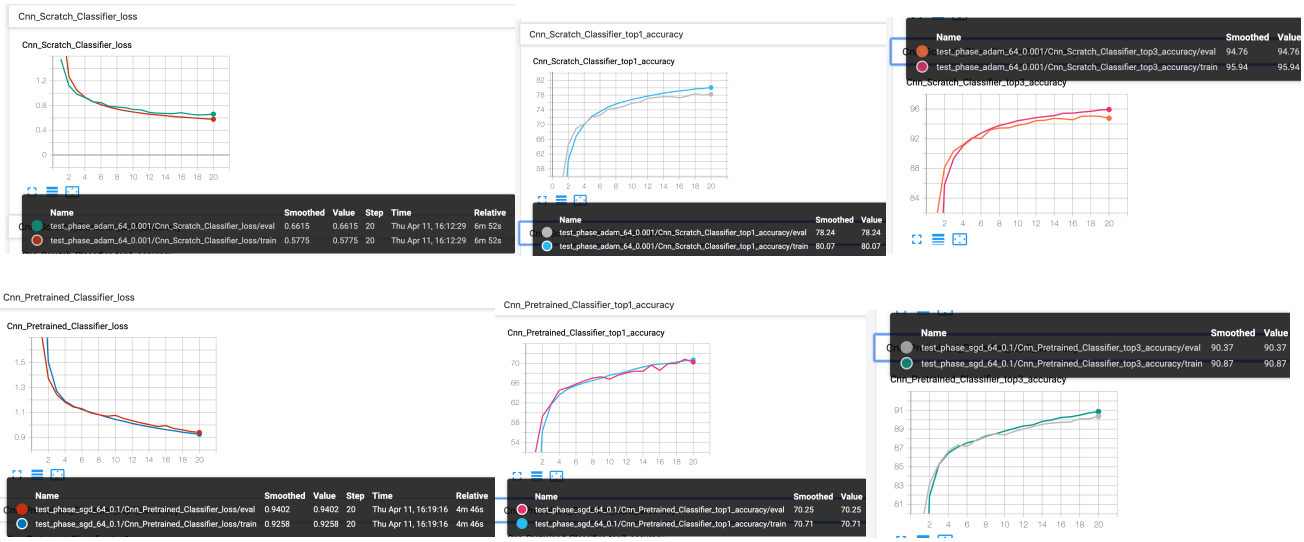




2. Number of epochs

For both of models, I originally set the total number of epochs to be 10 and the way of learning from scratch performs better than the other (not very significantly).

Then I set the epochs to be 20, the difference becomes obvious:



The results show that with the increase of the number of epochs in rational range, the performance of the models can improve, and CNN classifier from scratch shows better upper limit than the other one.

Another issue is that the model from CAE encoder ‘converges’ faster than the other one. I think this is because the fixed weights of encoder give the model less variation space, which will lead to the model’s getting faster to its own optimal setting.

3. Performance Metrics

From my test results, the performance of CNN classifier from scratch is better than the one from pretrained model. (Using 10 epochs as standard), the results are:

Scratch:

adam_64_0.001	1st	2nd	3rd	4th	5th	average	std
Top1 acc	75.7913	78.2767	78.4466	76.6578	74.2282	76.68012	1.765781
Top3 acc	93.9757	94.9684	95.0485	94.5461	92.7354	94.25482	0.95006756
loss	0.7316	0.657	0.6503	0.6875	0.8078	0.70684	0.06492459

Pretrained:

sgd_64_0.1	1st	2nd	3rd	4th	5th	average	std
Top1 acc	67.1019	68.466	67.8617	67.6602	67.9199	67.80194	0.49208935
Top3 acc	88.3495	89.1311	88.7961	88.3447	88.7621	88.6767	0.33365878
loss	1.0621	1.0202	1.0344	1.062	1.0417	1.04408	0.01813469

To conclude, the scratch model has better performance but takes more time. The pretrained model takes less time but do not perform as well as the scratch one

Part4: CAE with Pretrained Encoder

1. My CAE class:

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

   self.cnn_layers = torch.load('pretrained_encoder.pt', map_location = 'cpu')['model']

   # linear layers transforms flattened image features into logits before the softmax layer
   self.decoder = nn.Sequential(

     nn.ConvTranspose2d(in_channels=32, out_channels=16, kernel_size=3, stride=1, padding=0,output_padding=0, groups=1, bias=True, dilation=1),
     nn.ReLU(),

     nn.ConvTranspose2d(in_channels=16, out_channels=8, kernel_size=3, stride=1, padding=0,output_padding=0, groups=1, bias=True, dilation=1),
     nn.ReLU(),

     nn.ConvTranspose2d(in_channels=8, out_channels=8, kernel_size=3, stride=2, padding=0,output_padding=0, groups=1, bias=True, dilation=1),
     nn.ReLU(),

     nn.ConvTranspose2d(in_channels=8, out_channels=4, kernel_size=3, stride=1, padding=0,output_padding=0, groups=1, bias=True, dilation=1),
     nn.ReLU(),

     nn.ConvTranspose2d(in_channels=4, out_channels=1, kernel_size=4, stride=2, padding=0,output_padding=0, groups=1, bias=True, dilation=1),

     # As required, just after the last convolution layer, logistic function should be used
     nn.Sigmoid()
   )

   self.softmax = nn.Softmax(dim=1)
   # As required, here i need to use the MSE loss as the loss function
   self.loss_fn = nn.MSELoss(reduction='sum') # will be divided by batch size

 def forward(self, in_data):
   img_features = self.cnn_layers(in_data).view(in_data.size(0), 32,1,1) # in_data.size(0) == batch_size
   logits = self.decoder(img_features)
   return logits

 def loss(self, logits, labels):
   preds = self.softmax(logits) # size (batch_size, 10)
   return self.loss_fn(preds, labels) / logits.size(0) # divided by batch_size
```

Still, the encoder is loaded from the pt. file, and the decoder is designed as required.

Notice that the loss function is MSE for this part. I use the MSELoss provided by torch.

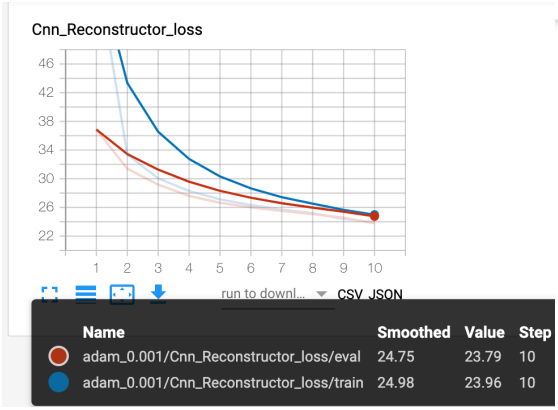
Then is the train function: Notice that ‘labels’ in dataset is not useful in this part.

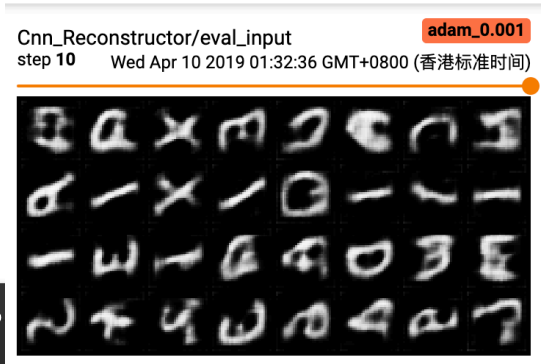
```
def train_reconstructor(model, loaders, optimizer, writer, n_epochs, ckpt_path, device='cpu'):
  def run_epoch(train_or_eval):
    epoch_loss = 0.

    for i, batch in enumerate(loaders[train_or_eval], 1):
      in_data, labels = batch
      in_data, labels = in_data.to(device), labels.to(device)

      if train_or_eval == 'train':
        optimizer.zero_grad()

      logits = model(in_data)

      batch_loss = model.loss(logits, in_data)

      epoch_loss += batch_loss

      if train_or_eval == 'train':
        batch_loss.backward()
        optimizer.step()

    epoch_loss /= i
    print('Loss: %s' %epoch_loss)
    print()

    losses[train_or_eval] = epoch_loss

    if writer is None:
      print('epoch %d %s loss %.4f acc %.4f' % (epoch, train_or_eval, epoch_loss, epoch_acc))
    elif train_or_eval == 'eval':
      writer.add_scalars('%s_loss' % model.__class__.__name__,# CnnClassifier or FcClassifier
                         tag_scalar_dict={'train': losses['train'],
                                          'eval': losses['eval']},
                         global_step=epoch)

      # For instructional purpose, add images here, just the last in_data
      if epoch % 10 == 0:
        if len(logits.size()) == 2: # when it is flattened, reshape it
          logits = logits.view(-1, 1, 28, 28)

        img_grid = make_grid(logits.to('cpu'))
        writer.add_image('%s/reconstruction' % model.__class__.__name__, img_grid, epoch)

  # main statements
  losses = dict()
```

```
for epoch in range(1, n_epochs+1):
  print('Epoch %s :' %epoch)
  print('Train:')
  run_epoch('train')
  print('Eval:')
  run_epoch('eval')

  # For instructional purpose, show how to save checkpoints
  if ckpt_path is not None:
    torch.save({
      'model_state_dict': model.state_dict(),
      'optimizer_state_dict': optimizer.state_dict(),
      'epoch': epoch,
      'losses': losses,
    }, '%s/%d.pt' % (ckpt_path, epoch))
```

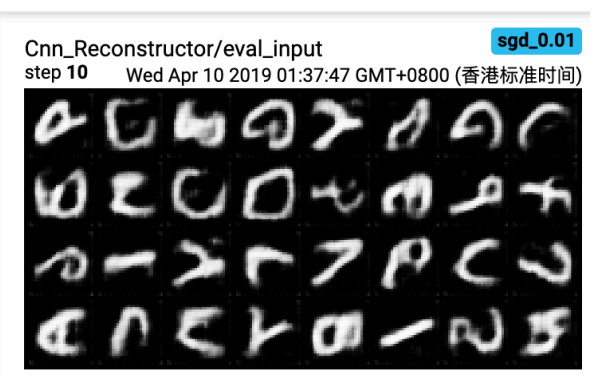
2. With the functions above, I can do the training phase with optional parameter sets. Here are the relevant results shown in TensorBoardX:

Adam_0.001:

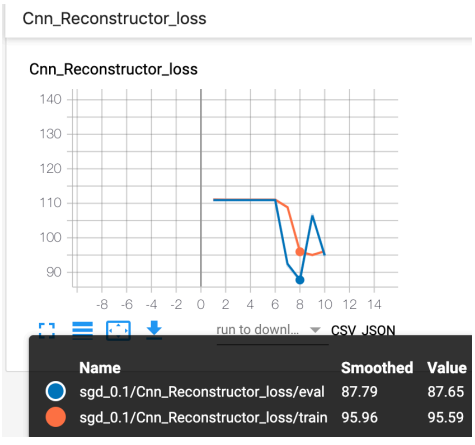


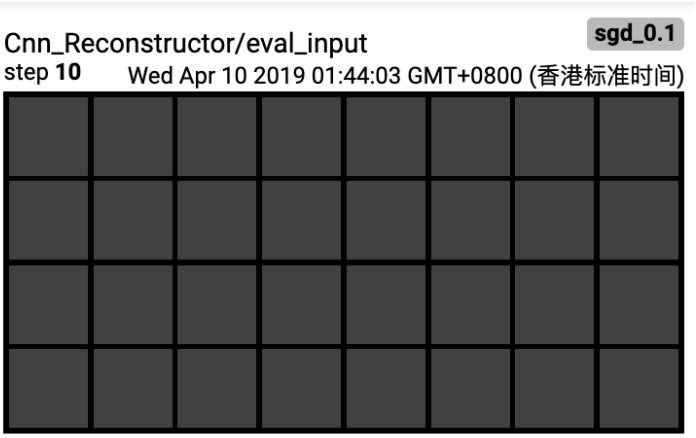


Sgd_0.01:

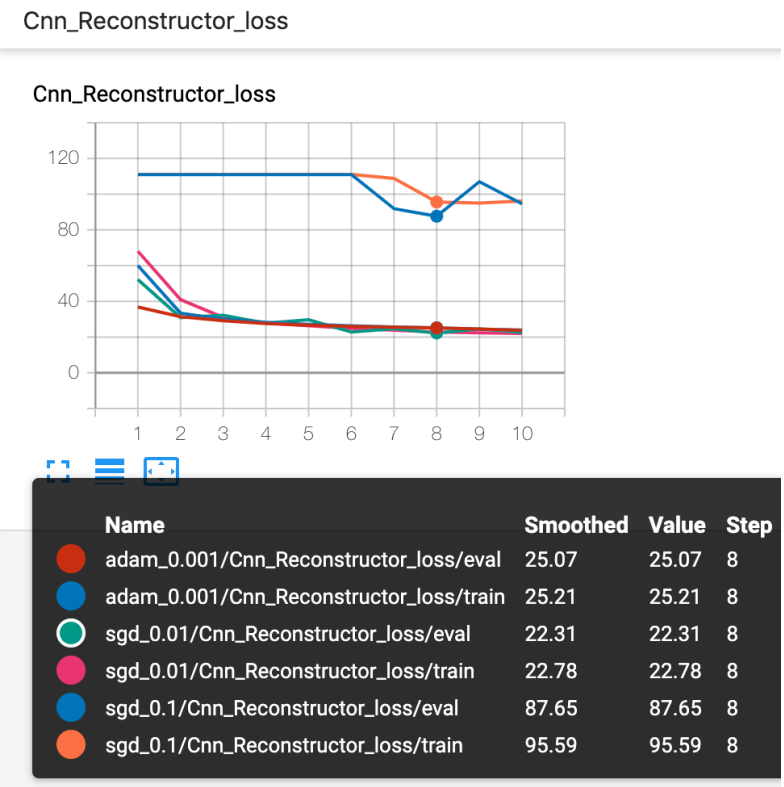


Sgd_0.1:





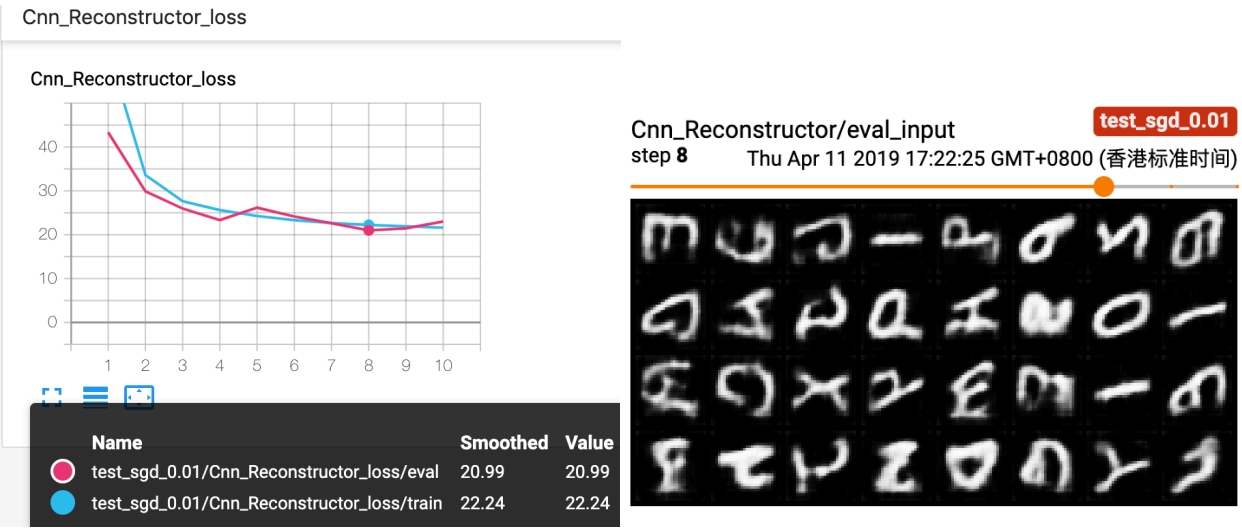
Overall:



I will choose {sgd_0.01} as the parameter set in the following testing phase.

3. Testing Phase:

   One of the five times' results:



The final results:

sgd_0.01	1st	2nd	3rd	4th	5th	average	std
loss	23.65	21.2252	20.99	21.2396	21.3541	21.69178	1.10264593