Lab 3 Exercises for COMP 691 (Deep Learning)

In this lab we will go over over basic stochastic optimization and how to use it in PyTorch.

- You will use the neural network you setup in lab 2 Exercise 2.
- Learn about parameter initialization.
- Learn about cross entropy loss.
- Train the neural network using mini-batch SGD w/o momentum.

Start by making a **copy** of this notebook in your Google Colab.

A good source for understanding leaf tensors (https://stackoverflow.com/questions/65301875/how-to-understand-creating-leaf-tensors-in-pytorch#:~:text=When%20a%20tensor%20is%20first,not%20a%20leaf%20node%20anymore.&text=requires_grad_()%20is%20not%20an,cuda()%20or%20c

Exercise 1: Loading the dataset

Setup the MNIST dataloaders for both the training (as well test) set as in Lab 2, Exercise 1. You do not need to iterate through the dataloaders, these will be used in the rest of the lab.

In [18]:
```
# This cell does not require outputing anything but is setup for subsequent cells
import torch
from torch import nn
from torchvision import transforms,datasets
from torch.utils import data
from torch.nn import functional as F
from matplotlib import pyplot as plt
import math

transform=transforms.Compose([
        transforms.ToTensor()
        ])

train_batch_size = 256
test_batch_size = 256

# load MNIST train and test sets
mnist_train = datasets.MNIST(root='.',
                             train=True,
                             download=True,
                              transform=transform)
mnist_test = datasets.MNIST(root='.',
                            train=False,
                            download=True,
                            transform=transform)

# initialize dataloaders for MNIST train and test sets
train_dataloader = data.DataLoader(mnist_train,batch_size=train_batch_size,drop_last=True)
test_dataloader = data.DataLoader(mnist_test,batch_size=test_batch_size,drop_last=True)
```

Exercise 2: Setting up the neural network

Consider your work from Lab 2, Exercise 2. Modify the neural network as follows.

- The hidden outputs should have size 100 in both hidden layers and the activations should be `relu` .
- Modify the final layer to output 10 values instead of 1. This means that instead of having a single scalar value, you will have 10 output classes!

Your code should implement

$$f(x) = W_2\rho(W_1\rho(W_0x + b_0) + b_1) + b_2$$

with $f : R^{786} -> R^{10}$ and $\rho =$ `relu`

Initialize the weights using a variant of xavier intialization

$$w_{ij} \sim \mathrm{N}\left(0, \frac{1}{\sqrt{n_i}}\right)$$

where n_i is the size of the layer input. Initialize the biases as 0.

Write a helper function to perform this initialization for subsequent parts of this lab.

In [19]:
```
import torch
import os
# set your target device
device = 'cuda' if torch.cuda.is_available() else 'cpu'

## Initialize and track the parameters using a list or dictionary
h1 = 50
h2 = 50
param_dict = {
    "W0": torch.rand(784, h1) * 2 -1,
    "b0": torch.zeros(h1),
    "W1": torch.rand(h1, h2) * 2 -1,
    "b1": torch.zeros(h2),
    "W2": torch.rand(h2, 10) * 2 -1,
    "b2": torch.zeros(10),
    }



## Define the network
def my_nn(input, param_dict):
    """Performs a single forward pass of a Neural Network with the given
    parameters in param_dict.

    Args:
        input (torch.tensor): Batch of images of shape (B, H, W), where B is
            the number of input samples,and H and W are the image height and
            width respectively.
        param_dict (dict of torch.tensor): Dictionary containing the parameters
            of the neural network. Expects dictionary keys to be of format
            "W#" and "b#" where # is the layer number.

    Returns:
        torch.tensor: Neural network output of shape (B, )
    """
    #Reshape the input image from HxW to a flat vector
    x = input.view(-1 , 28*28) #256*28*28
    #Your code here

    relu = torch.nn.ReLU()
    x = torch.matmul(x, param_dict['W0']) + param_dict['b0']
    x = relu(x)

    x = torch.matmul(x, param_dict['W1']) + param_dict['b1']
    x = relu(x)

    x = torch.matmul(x, param_dict['W2']) + param_dict['b2']

    return x

os.environ['CUDA_LAUNCH_BLOCKING'] = '1'

def initialize_nn(param_dict):
    """Takes a dictionary with existing torch tensors
        and re-initializes them using xavier initialization
    """

    # make sure that your parameters are on the chosen device and that they require gradients!
    for name, param in param_dict.items():
        param_dict[name] = torch.zeros_like(param)
        if "W" in name:
            param_dict[name] = param_dict[name].normal_(0,1/math.sqrt(param.size(0)))
        param_dict[name] = param_dict[name].to(device)
        param_dict[name].requires_grad = True

initialize_nn(param_dict)
```

Now, we will evaluate the cross entropy loss and average accuracy on this randomly initialized dataset.

Use the `torch.nn.functional.cross_entropy()` function to compute the loss $\frac{1}{N}\sum_i^N \text{CrossEntropy}(f(x_i), y_i)$ and accuracy.

Your accuracy should be close to 10% as the network has random weights and no training has happened yet!

In [20]:
```
cuda = torch.cuda.is_available()


def test(test_dataloader):
    loss = 0
    accuracy = 0
    for data, targets in test_dataloader:
        #move to GPU if available
        if cuda:
            data, targets= data.to('cuda'), target.to('cuda')

        #compute model output
        output = my_nn(data, param_dict)
        #comptue accuracy for minibatch
        prediction = output.argmax(1)
        accuracy += sum(prediction == targets).item()/test_batch_size
        #compute loss for minibatch
        loss += F.cross_entropy(output,targets).item()
    #aggregate loss and accuracy for all test data
    print(f'test loss: {loss/len(test_dataloader):4f}')
    print(f'test accuracy: {accuracy/len(test_dataloader)*100:.2f}%')
test(test_dataloader)
```

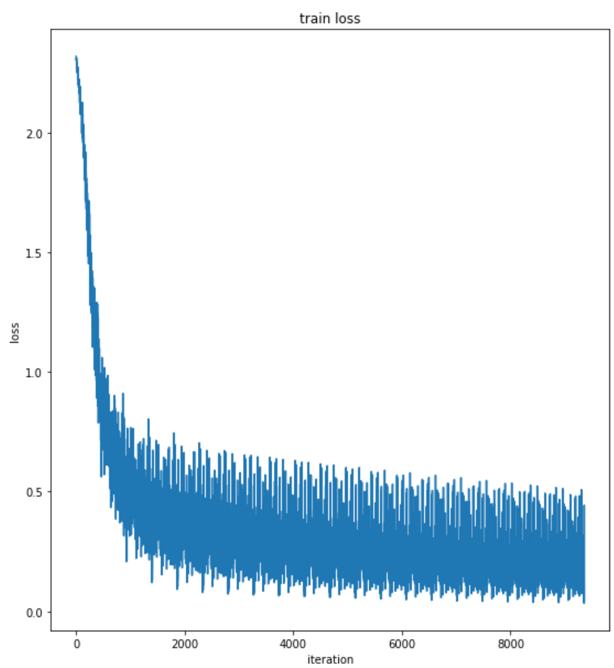
```
test loss: 2.328130
test accuracy: 9.04%
```

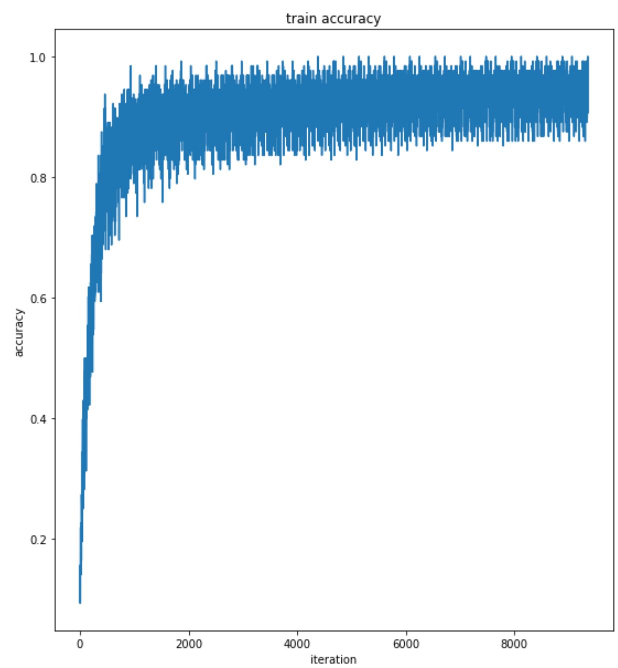
Exercise 3: Training using mini-batch SGD from scratch

Without using the `torch.optim` package implement from scratch mini-batch Stochastic Gradient Descent training to minimize the loss $\frac{1}{N}\sum_i^N \text{CrossEntropy}(f(x_i), y_i)$ over the MNIST dataset.

- Use a minibatch size of 128 and a learning rate of 0.01. Run the training for 20 epochs.
- You will PyTorch's autograd features (e.g. `.backward()`) to obtain the gradients given each mini-batch at each parameter. Modify the existing parameters based on the obtained gradient using the SGD update rule, e.g. $w = w - \alpha * \frac{\partial loss}{\partial w}$.
- Store the losses and training of each minibatch and plot each of these (with iterations (not epochs) as the x-axis). You can optionally smooth out these plots over 20-100 iteration window of your choosing to make them cleaner to read.
- Compute and report the final test accuracy as well at the end of the 20 epochs.

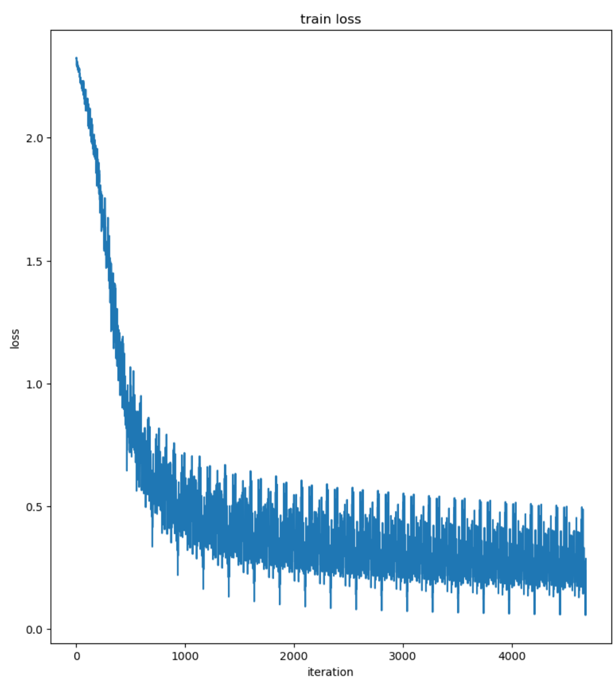
You should end up with 2 plots and a final test accuracy that looks something like this

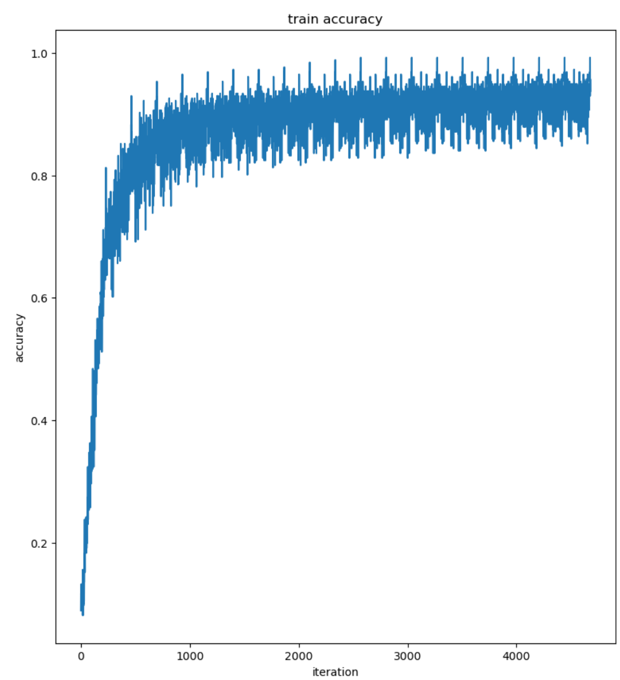




In [21]:
```
import torch


train_losses = [] # use to append the avg loss for each minibatch
train_accs = [] # use to append the avg acc of minibatch

alpha=0.01


for epoch in range(20):
    for data,target in train_dataloader: #Iterate over dataset
        #Compute the gradient for the minibatch
        data, target = data.to(device), target.to(device)
        output = my_nn(data,param_dict)

        loss = F.cross_entropy(output,target)
        loss.backward()

        train_losses.append(loss.item())
        prediction = output.argmax(1)
        train_accs.append(sum(prediction == target).item()/train_batch_size)
        for (name,param) in param_dict.items():
            with torch.no_grad():
                #Update the model parameters by w_t-alpha*grad
                param_dict[name] -= alpha * param_dict[name].grad
            if param.grad is not None:
                #The following code will clear the gradient buffers for the next iteration
                param.grad.detach_()
                param.grad.zero_()

    #Update loss and acc tracking

#Plot the train loss and acc
fig,axs = plt.subplots(1,2,figsize=(20,10))
axs[0].plot(train_losses)
axs[0].set_title('train loss')
axs[0].set_xlabel('iteration')
axs[0].set_ylabel('loss')
axs[1].plot(train_accs)
axs[1].set_title('train accuracy')
axs[1].set_xlabel('iteration')
axs[1].set_ylabel('accuracy')
#Evaluate on the test set
test(test_dataloader)
```

```
test loss: 0.257728
test accuracy: 92.49%
```





Exercise 4: Training using the pre-built torch.optim.SGD

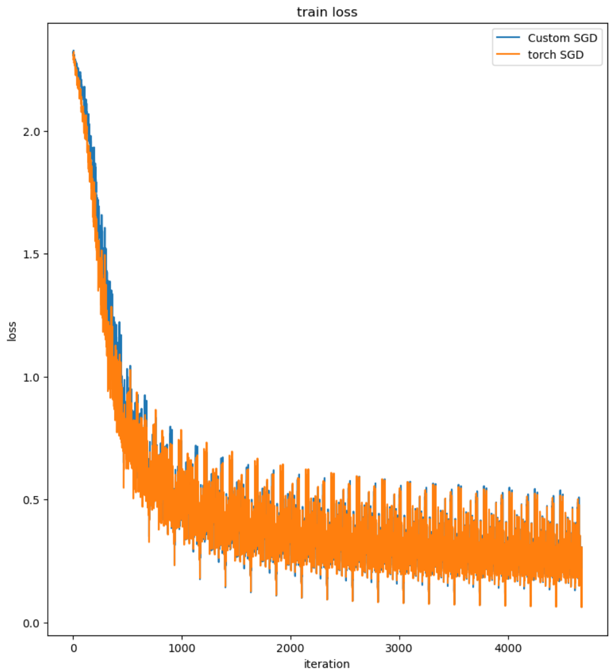
Repeat Exercise 3 but now using the package `torch.optim.SGD` to perform SGD:

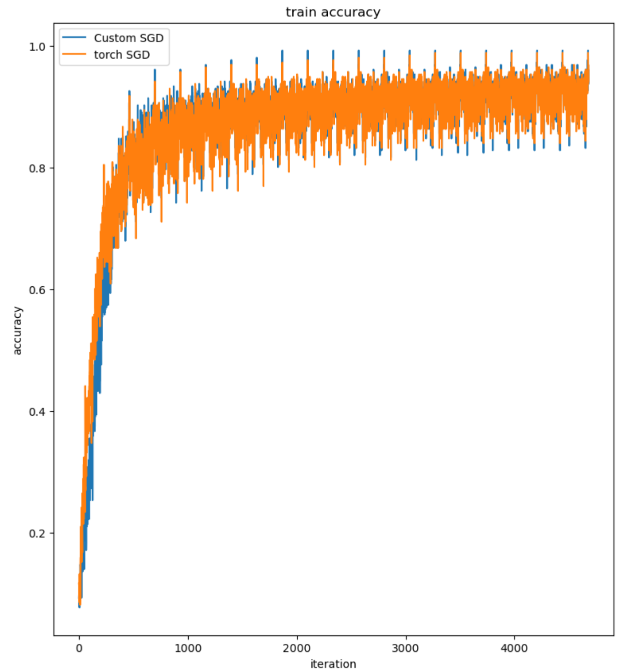
https://pytorch.org/docs/stable/optim.html?highlight=torch%20optim%20sgd#torch.optim.SGD (https://pytorch.org/docs/stable/optim.html?highlight=torch%20optim%20sgd#torch.optim.SGD)

Plot the same training curves and accuracy curves. Check that your learning curves are similar to those in Exercise 3. You can for example overlay them or plot them side by side with subplot.

In [25]:
```
#Answer for SGD
import torch
#Make sure to reinitialize your network to random before starting training
initialize_nn(param_dict)

#optim.SGD takes a list of parameters which you can get from your dictionary as follows
parameter_list = param_dict.values()

optimizer = torch.optim.SGD(parameter_list, lr=alpha)

def train(train_dataloader,optimizer):
    train_losses = [] # use to append the avg loss for each minibatch
    train_accs = [] # use to append the avg acc of minibatch
    #iterate over the minibatches
    for epoch in range(20):
        for data,target in train_dataloader:
            #move to GPU if available
            data, target = data.to(device), target.to(device)

            # set gradients to zero
            optimizer.zero_grad()
            #compute model output
            output = my_nn(data,param_dict)
            #compute the gradient for the minibatch
            loss = F.cross_entropy(output,target)
            loss.backward()
            # update weights
            optimizer.step()
            # track training loss and accuracy
            train_losses.append(loss.item())
            prediction = output.argmax(1)
            train_accs.append(sum(prediction == target).item()/train_batch_size)
            #update the model parameters by w_t-alpha*\grad_{w_t}(f(w_t))
        # print training progress
        print(f' epoch {epoch}: train loss {sum(train_losses)/(len(train_dataloader)*(epoch+1)):.4f}- train accurac
    return train_losses,train_accs

train_losses_torch,train_accs_torch = train(train_dataloader,optimizer)
#Plot the train loss and acc
fig,axs = plt.subplots(1,2,figsize=(20,10))
axs[0].plot(train_losses,label='Custom SGD')
axs[0].plot(train_losses_torch,label='torch SGD')
axs[0].set_title('train loss')
axs[0].set_xlabel('iteration')
axs[0].set_ylabel('loss')
axs[0].legend()
axs[1].plot(train_accs,label='Custom SGD')
axs[1].plot(train_accs_torch,label='torch SGD')
axs[1].set_title('train accuracy')
axs[1].set_xlabel('iteration')
axs[1].set_ylabel('accuracy')
axs[1].legend()
#Evaluate on the test set
test(test_dataloader)
```

```
 epoch 0: train loss 1.9514- train accuracy 44.42%
 epoch 1: train loss 1.5171- train accuracy 59.78%
 epoch 2: train loss 1.2413- train accuracy 67.10%
 epoch 3: train loss 1.0691- train accuracy 71.49%
 epoch 4: train loss 0.9514- train accuracy 74.49%
 epoch 5: train loss 0.8653- train accuracy 76.69%
 epoch 6: train loss 0.7992- train accuracy 78.37%
 epoch 7: train loss 0.7466- train accuracy 79.73%
 epoch 8: train loss 0.7037- train accuracy 80.83%
 epoch 9: train loss 0.6678- train accuracy 81.75%
 epoch 10: train loss 0.6374- train accuracy 82.53%
 epoch 11: train loss 0.6112- train accuracy 83.20%
 epoch 12: train loss 0.5883- train accuracy 83.79%
 epoch 13: train loss 0.5682- train accuracy 84.31%
 epoch 14: train loss 0.5502- train accuracy 84.77%
 epoch 15: train loss 0.5341- train accuracy 85.19%
 epoch 16: train loss 0.5196- train accuracy 85.57%
 epoch 17: train loss 0.5063- train accuracy 85.92%
 epoch 18: train loss 0.4941- train accuracy 86.24%
 epoch 19: train loss 0.4830- train accuracy 86.53%
test loss: 0.260503
test accuracy: 92.38%
```





Optional: Training using mini-batch SGD with momentum

Modify the code from Exercises 3 and 4 to perform minibatch SGD with momentum. Use a momentum of $\mu = 0.9$ and learning rate $\alpha = 0.01$. Use the following formulation of momentum:

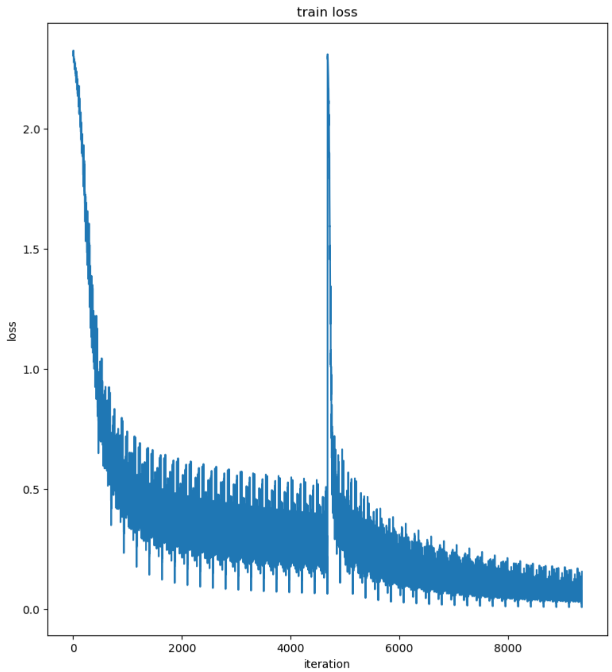
- $g = \nabla_w CE(f(w_t, X), Y)$: gradient estimate with mini-batch
- $v_{t+1} = \mu * v_t + g$
- $w_{t+1} = w - \alpha * v_{t+1}$

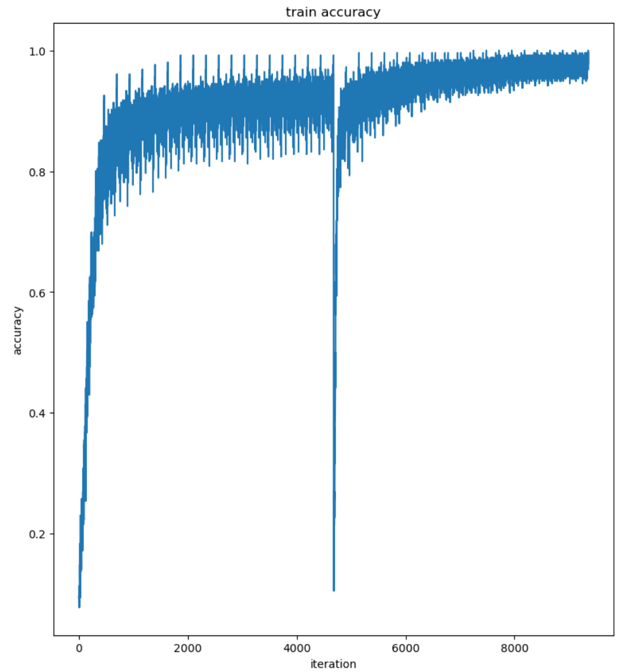
Obtain the same plots as before and a final test accuracy.

In [26]:

```
# Reinitialize your network to random!
initialize_nn(param_dict)

losses = [] # use to append the avg loss for each minibatch
train_acc = [] # use to append the avg acc of minibatch

alpha=0.01
mu=0.9

# initialize velocity for every parameter
v={}
for (name,param) in param_dict.items():
    v[name] = torch.zeros_like(param)
for epoch in range(20):
    for data,target in train_dataloader:
        #move to GPU if available
        #move to GPU if available
        data, target = data.to(device), target.to(device)

        #compute model output
        output = my_nn(data,param_dict)
        #compute the gradient for the minibatch
        loss = F.cross_entropy(output,target)
        loss.backward()
        # track training loss and accuracy
        train_losses.append(loss.item())
        prediction = output.argmax(1)
        train_accs.append(sum(prediction == target).item()/train_batch_size)
        #update the model parameters by w_t-alpha*\grad_{w_t}(f(w_t))
        for (name,param) in param_dict.items():
            with torch.no_grad():
                v[name] = mu*v[name] + param_dict[name].grad
                param_dict[name] -= alpha * v[name]
            if param.grad is not None:
                param_dict[name].grad.detach_()
                param_dict[name].grad.zero_()

    #Update loss and acc tracking

#Plot the train loss and acc
#Plot the train loss and acc
fig,axs = plt.subplots(1,2,figsize=(20,10))
axs[0].plot(train_losses)
axs[0].set_title('train loss')
axs[0].set_xlabel('iteration')
axs[0].set_ylabel('loss')
axs[1].plot(train_accs)
axs[1].set_title('train accuracy')
axs[1].set_xlabel('iteration')
axs[1].set_ylabel('accuracy')
#Evaluate on the test set
test(test_dataloader)
# save train loss and accuracy for further comparison
train_loss_mom = train_losses
train_accs_mom = train_accs
```

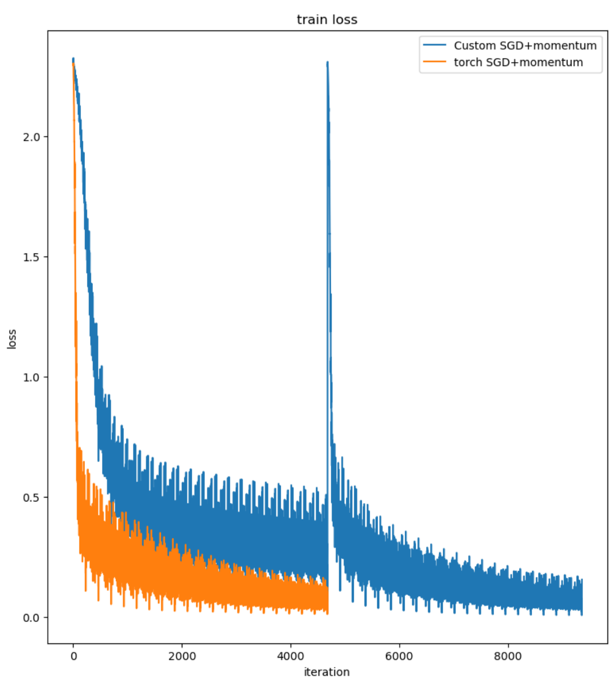
```
test loss: 0.099428
test accuracy: 96.99%
```

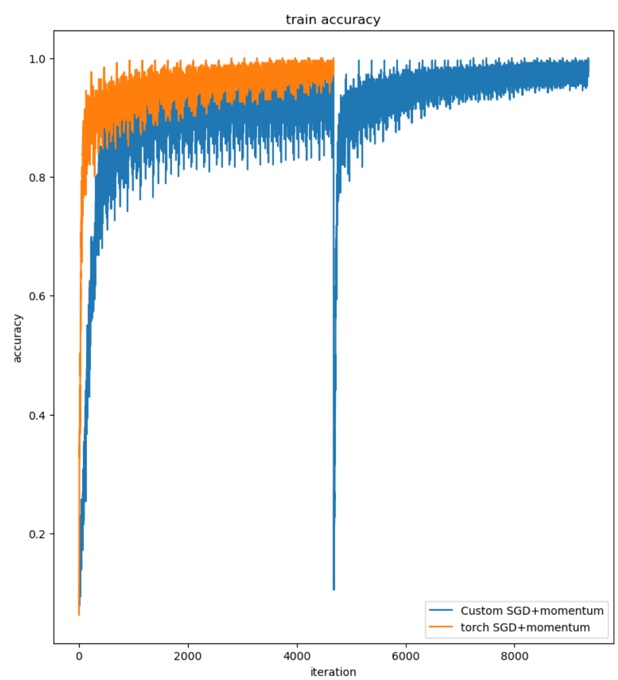

train loss
loss
iteration
0.0
0.5
1.0
1.5
2.0
0
2000
4000
6000
8000


train accuracy
accuracy
iteration
0.2
0.4
0.6
0.8
1.0
0
2000
4000
6000
8000

In [27]:
```
# Answer for SGD+momentum
initialize_nn(param_dict)

#optim.SGD takes a list of parameters which you can get from your dictionary as follows
parameter_list = param_dict.values()

optimizer = torch.optim.SGD(parameter_list, lr=alpha, momentum=mu)

train_losses , train_accs = train(train_dataloader,optimizer)

#Plot the train loss and acc
fig,axs = plt.subplots(1,2,figsize=(20,10))
axs[0].plot(train_loss_mom,label='Custom SGD+momentum')
axs[0].plot(train_losses,label='torch SGD+momentum')
axs[0].set_title('train loss')
axs[0].set_xlabel('iteration')
axs[0].set_ylabel('loss')
axs[0].legend()
axs[1].plot(train_accs_mom,label='Custom SGD+momentum')
axs[1].plot(train_accs,label='torch SGD+momentum')
axs[1].set_title('train accuracy')
axs[1].set_xlabel('iteration')
axs[1].set_ylabel('accuracy')
axs[1].legend()
#Evaluate on the test set
test(test_dataloader)
```

```
 epoch 0: train loss 0.7873- train accuracy 77.28%
 epoch 1: train loss 0.5534- train accuracy 84.01%
 epoch 2: train loss 0.4584- train accuracy 86.75%
 epoch 3: train loss 0.4025- train accuracy 88.37%
 epoch 4: train loss 0.3638- train accuracy 89.49%
 epoch 5: train loss 0.3344- train accuracy 90.34%
 epoch 6: train loss 0.3109- train accuracy 91.03%
 epoch 7: train loss 0.2914- train accuracy 91.59%
 epoch 8: train loss 0.2749- train accuracy 92.07%
 epoch 9: train loss 0.2607- train accuracy 92.48%
 epoch 10: train loss 0.2482- train accuracy 92.84%
 epoch 11: train loss 0.2371- train accuracy 93.16%
 epoch 12: train loss 0.2272- train accuracy 93.44%
 epoch 13: train loss 0.2183- train accuracy 93.70%
 epoch 14: train loss 0.2101- train accuracy 93.93%
 epoch 15: train loss 0.2026- train accuracy 94.15%
 epoch 16: train loss 0.1958- train accuracy 94.35%
 epoch 17: train loss 0.1894- train accuracy 94.53%
 epoch 18: train loss 0.1835- train accuracy 94.70%
 epoch 19: train loss 0.1780- train accuracy 94.86%
test loss: 0.099249
test accuracy: 96.73%
```





In [ ]: