40216952_2023_Lab4_Ex

February 10, 2023

1 Lab 4 - Batch Normalization

In this lab has the following goals: - Implement functional and module based batch normalization layer. - Understand the subtleties regarding batchnorm usage, particularly avoiding statistic computation in the test set. - Introduce the use of `register_buffer` in `torch.nn.Module`. - Understand the `.eval()` and `.train()` methods of `torch.nn.Module` and what these do.

Note: It is recommended to run the lab mini-experiments on GPU.

IMPORTANT: For submission you are **only** required to complete **Part 1**: Functional Batch Normalization.

1.1 0 Initialization

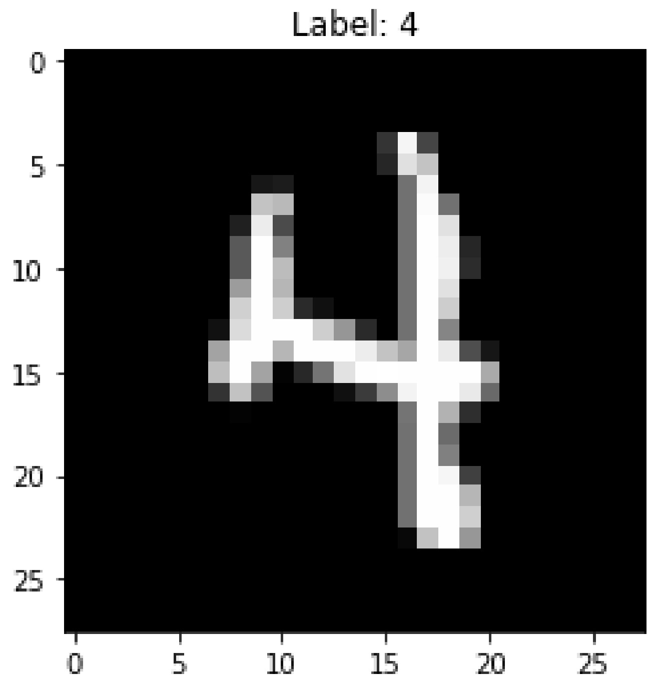
Run the code cells below to initialize the train and test loaders of the MNIST dataset and visualize one of the MNIST samples.

[ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import torch
from torchvision import datasets,transforms


# Initialize train and test datasets
train_set = datasets.MNIST('../data',
                           train=True,
                           download=True,
                           transform=transforms.ToTensor())
test_set = datasets.MNIST('../data',
                          train=False,
                          download=True,
                          transform=transforms.ToTensor())

# Initialize train and test data loaders
train_loader = torch.utils.data.DataLoader(train_set,
                                           batch_size=256,
                                           shuffle=True,
                                           drop_last=True)
test_loader = torch.utils.data.DataLoader(test_set,
```

```
                                          batch_size=256,
                                          shuffle=True,
                                          drop_last=True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
../data/MNIST/raw/train-images-idx3-ubyte.gz

  0%|          | 0/9912422 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/train-images-idx3-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
../data/MNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/28881 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/train-labels-idx1-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
../data/MNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/1648877 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/t10k-images-idx3-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
../data/MNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/4542 [00:00<?, ?it/s]

Extracting ../data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/MNIST/raw
```

[ ]:
```
# Visualize a sample from MNIST
X_train_samples, y_train_samples = next(iter(train_loader))
plt.title(f'Label: {y_train_samples[0]}')
plt.imshow((X_train_samples[0].squeeze(0)).numpy(), cmap='gray');
```



1.2 Exercise 1: Functional Batch Normalization

1.2.1 1.1 Batch Normalization Function

Implement a function that performs batch normalization on a given `inputs` tensor of shape `(N, F)`, where `N` is the minibatch size and `F` is the number of features.

Note: Batch normalization performs differently at train and inference time: * `train`: During training, batch normalization standardizes the given inputs along the minibatch dimension (mean and standard deviation would be of shape `(F,)`) using the equation given below. The running average of the minibatch means and variances are updated during training using the equations on slide 30 of lecture 4. Learnable parameters β and γ shift and scale the distribution after standardization. ϵ is a constant and will be set to 0.001. * `eval`: During evaluation (inference), batch normalization uses the running average of the means and standard deviations which were computed during training for normalization.

Implement a functional batch normalization layer with the differentiable affine parameters γ and β. The batch normalization layer has the following formulation:

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

You will need to create an additional set of variables to track and update the statistics (`toy_stats_dict`). Note that the statistics are updated outside of backpropagation. For the momentum rate of batchnorm statistics use `0.1`.

Your function is then checked in train mode with 100 sample random values $\sim \mathcal{N}(50, 10)$ (so shape would be `(100, 1)`. The correct printed output should be (very close to):

```
Training Samples
Before BN: mean tensor([54.8908]), var tensor([8.1866])
After BN: mean tensor([-1.3208e-06], grad_fn=<MeanBackward1>), var tensor([0.9999], grad_fn=<Va
```

Note: To get these exact same values, your device would need to be set to cpu.

[ ]:
```python
# Set seed
torch.manual_seed(691)

# Number of features
train_size = 500
test_size = 1
num_features = 1

# Generates toy train features for evaluating your function down below
toy_train_features = (torch.rand(train_size, num_features) * 10) + 50

### TODO: Initialize the `running_mean` and `running_var` variables
### with 0 and 1 values respectively.
toy_stats_dict = {
    "running_mean": torch.zeros(num_features),
    "running_var": torch.ones(num_features),
}

### TODO: Initialize the learnable parameters `beta` and `gamma`
### with 0 and 1 values respectively.
beta = torch.zeros(num_features, requires_grad=True, dtype=torch.float32)
gamma = torch.ones(num_features, requires_grad=True, dtype=torch.float32)

def batchnorm(inputs, beta, gamma, stats_dict, train=True, eps=0.001,␣
 ↪momentum=0.1):
    r"""Performs batch normalization for a single layer of inputs. If in train
    mode, will update the stats_dict dictionary with running mean and variance
    values.

    Args:
        inputs (torch.tensor): Batch of inputs of shape (N, F), where N is
            the minibatch size, and F is the number of features.
        beta (torch.tensor): Batch normalization beta variable of shape (F,).
        gamma (torch.tensor): Batch normalization gamma variable of shape (F,).
        stats_dict (dict of torch.tensor): Dictionary containing the running
            mean and variance. Expects dictionary to contain keys 'running_mean'
            and 'running_var', with values being `torch.tensor`s of shape (F,).
        train (bool): Determines whether batch norm is in train mode or not.
            Default: True
```

```
        eps (float): Constant for numeric stability.
        momentum (float): The momentum value for updating the running mean and
            variance during training.

    Returns:
        torch.tensor: Batch normalized inputs, of shape (N, F)
    """
    ### TODO: Fill out this function
    # 1. Calculate the mean and variance of the inputs
    run_mean = stats_dict["running_mean"]
    run_var = stats_dict["running_var"]

    if train:
        mean = inputs.mean(0)
        var = inputs.var(0)

        run_mean = momentum * mean + (1 - momentum) * run_mean
        run_var = momentum * var + (1 - momentum) * run_var

        stats_dict["running_mean"] = run_mean
        stats_dict["running_var"] = run_var
        return gamma * (inputs - mean) / torch.sqrt(var + eps) + beta
    else:
        return gamma * (inputs - run_mean) / torch.sqrt(run_var + eps) + beta


# run batchnorm on toy train features
bn_out_train = batchnorm(toy_train_features, beta, gamma, toy_stats_dict)

# print results
print("Training Samples")
print(f"Before BN: mean {toy_train_features.mean(0)}, var {toy_train_features.
 ↪var(0)}")
print(f"After BN: mean {bn_out_train.mean(0)}, var {bn_out_train.var(0)}\n")
```

```
Training Samples
Before BN: mean tensor([54.8908]), var tensor([8.1866])
After BN: mean tensor([-1.3208e-06], grad_fn=<MeanBackward1>), var
tensor([0.9999], grad_fn=<VarBackward0>)
```

1.2.2 1.2 Setting up the Model Architecture

For the model architecture, you will use the 2 layer model from labs 2 & 3 (the one that doesnt use `nn.Module`). You will use the batchnorm function defined in part (1.1) at the 2 hidden layers of the network. Batch normalization is typically applied before the activation function!

Note: You will need 2 variables one for each layer to track the statistics, i.e., the running mean and variance.

Modify your intialization function that you implemented in lab 3. The function should do the following: - Initialize β's with zeros and γ's with ones. - Intialize the variables that contain the running mean and variance of each layer. - Intialize all parameters in the network (done in lab 2).

VERY IMPORTANT: Make sure that ALL the trainable parameters require gradient!

[ ]:
```
# Initialize model hiden layer sizes
h1_size = 50
h2_size = 50

### TODO: Initialize the beta and gamma parameters
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
beta0 = torch.zeros(h1_size, dtype=torch.float32)
gamma0 = torch.ones(h1_size, dtype=torch.float32)
beta1 = torch.zeros(h2_size, dtype=torch.float32)
gamma1 = torch.ones(h2_size, dtype=torch.float32)

# Intentional naive initialization (do not modify)
param_dict = {
    "W0": torch.rand(784, h1_size)*2-1,
    "b0": torch.rand(h1_size)*2-1,
    "beta0": beta0,
    "gamma0": gamma0,
    "W1": torch.rand(h1_size, h2_size)*2-1,
    "b1": torch.rand(h2_size)*2-1,
    "beta1": beta1,
    "gamma1": gamma1,
    "W2": torch.rand(h2_size,10)*2-1,
    "b2": torch.rand(10)*2-1,
}

for name, param in param_dict.items():
    param_dict[name] = param.to(device)
    param_dict[name].requires_grad = True

### TODO: Initialize the `running_mean` and `running_var` variables
### with 0s and 1s respectively.
l1_stats_dict = {
    "running_mean": torch.zeros(h1_size),
    "running_var": torch.ones(h1_size),
}
l2_stats_dict = {
    "running_mean": torch.zeros(h2_size),
    "running_var": torch.ones(h2_size),
}
layers_stats_list = [l1_stats_dict, l2_stats_dict]
```

```python
def my_nn(input, param_dict, layers_stats_list, train=True):
    r"""Performs a single forward pass of a 2 layer MLP with batch
    normalization using the given parameters in param_dict and the
    batch norm statistics in layers_stats_list.

    Args:
        input (torch.tensor): Batch of images of shape (N, H, W), where N 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#", "b#", "beta#" and "gamma#" where # is the layer number.
        layers_stats_list (list of dict of torch.tensor): List of dictionaries
            containing running means and variances for each layer. List size
            is equal to the number of hidden layers.
        train (bool): Determines whether batch norm is in train mode or not.
            Default: True

    Returns:
        torch.tensor: Neural network output of shape (N, 10)
    """
    x = input.view(-1, 28*28)

    # layer 1
    x = torch.relu_(x @ param_dict['W0'] + param_dict['b0'])

    ### TODO: use your complete batchnorm function
    x = batchnorm(x, param_dict['beta0'], param_dict['gamma0'],␣
 ↪layers_stats_list[0], train)

    # layer 2
    x = torch.relu_(x @ param_dict['W1'] + param_dict['b1'])

    ### TODO: use your complete batchnorm function
    x = batchnorm(x, param_dict['beta1'], param_dict['gamma1'],␣
 ↪layers_stats_list[1], train)

    # output
    x = x @ param_dict['W2'] + param_dict['b2']
    return x

def my_zero_grad(param_dict):
    r"""Zeros the gradients of the parameters in `param_dict`.

    Args:
        param_dict (dict of torch.tensor): Dictionary containing the parameters
```

```
            of the neural network. Expects dictionary keys to be of format
            "W#", "b#", "beta#" and "gamma#" where # is the layer number.
        layers_stats_list (list of dict of torch.tensor): List of dictionaries
            containing running means and variances for each layer. List size
            is equal to the number of hidden layers.

    Returns:
        None
    """
    for _, param in param_dict.items():
        if param.grad is not None:
            param.grad.detach_()
            param.grad.zero_()

def initialize_nn(param_dict, layers_stats_list):
    r"""Initializes the parameters in `param_dict` and resets the statistics
    in `layers_stats_list`.

    Args:
        param_dict (dict of torch.tensor): Dictionary containing the parameters
            of the neural network. Expects dictionary keys to be of format
            "W#", "b#", "beta#" and "gamma#" where # is the layer number.
        layers_stats_list (list of dict of torch.tensor): List of dictionaries
            containing running means and variances for each layer. List size
            is equal to the number of hidden layers.

    Returns:
        None
    """
    ### TODO: Fill out this function
    for name,param in param_dict.items():
        if "beta" in name:
            param_dict[name] = torch.zeros_like(param)
        elif "gamma" in name:
            param_dict[name] = torch.ones_like(param)
        else:
            param_dict[name] = torch.rand_like(param)*2-1

    for name, param in param_dict.items():
        param_dict[name] = param.to(device)
        param_dict[name].requires_grad = True

    for layer_stats in layers_stats_list:
        layer_stats["running_mean"] = torch.
↪zeros_like(layer_stats["running_mean"],device = device)
        layer_stats["running_var"] = torch.
↪ones_like(layer_stats["running_var"], device = device)
```

1.2.3 1.3 Training the Model

Train the model on the MNIST dataset with 20 epochs and `lr=0.01` with SGD and without momentum (as per lab 2). Since you are dealing with the MNIST dataset and you are required to perform multiclass classification, you will use the cross entropy loss during training (similar to lab2).

Plot the learning curves for training accuracy recorded every 50 iterations (smoothed output).

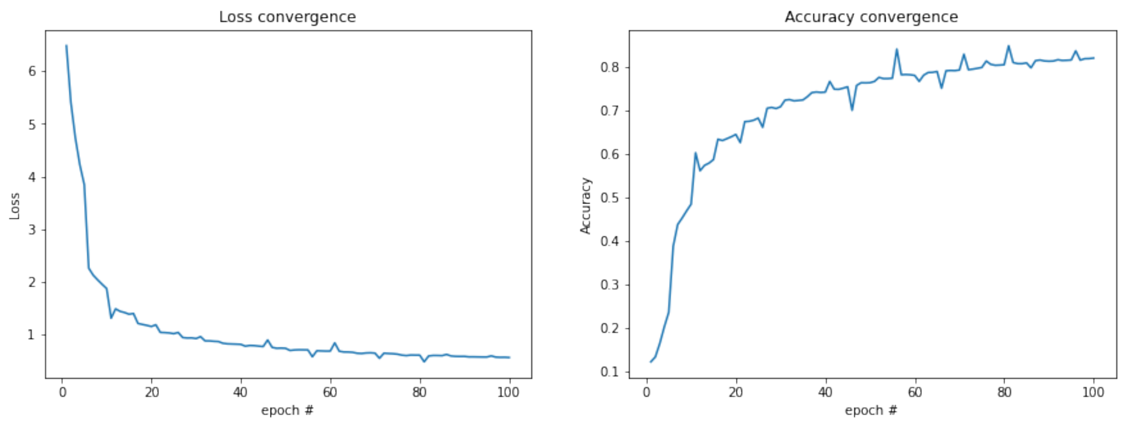
The first 5 epochs should have close values to the following output:

This is a sample output of how your plots should look like:

[ ]:
```python
from torch.optim import SGD

# training hyper_parameters
lr = 0.01
num_epochs = 20

### TODO: Initialize optimizer. You can use the SGD class from pytorch.
initialize_nn(param_dict, layers_stats_list)
optimizer = SGD(param_dict.values(), lr=lr)
train_track={
    "loss": [],
    "acc": [],
}

for epoch in range(num_epochs):
    total_train = 0
    total_loss = 0
    total_correct = 0
    for i, (data,label) in enumerate(train_loader):
        ### TODO: Train the network
        data = data.to(device, dtype=torch.float32)
        label = label.to(device, dtype=torch.long)

        # forward pass
        output = my_nn(data, param_dict, layers_stats_list, train=True)
        loss = torch.nn.functional.cross_entropy(output, label)

        # backward pass
        optimizer.zero_grad() #Can also use my_zero_grad(param_dict)
        loss.backward()
        optimizer.step()
```

```
        m_batch = data.shape[0]
        total_train += m_batch
        total_loss += loss.item()*m_batch
        total_correct += (output.argmax(dim=1) == label).sum().item()

        if i % 50 == 0:
            train_loss = total_loss/total_train
            train_acc = total_correct/total_train
            train_track["loss"].append(train_loss)
            train_track["acc"].append(train_acc)

    print(f"Epoch: {epoch+1}/{num_epochs}, Step: {i+1}/{len(train_loader)},␣
 ↪Loss: {train_loss:.4f}, Acc: {train_acc:.4f}")

fig, ax = plt.subplots(1, 2, figsize=(15, 5))
for ax, key in zip(ax, train_track.keys()):
    loss_history = train_track[key]
    ax.plot(np.arange(len(loss_history))+1, loss_history)
    ax.set_xlabel("epoch #")
    ylabel = "Loss" if key == "loss" else "Accuracy"
    ax.set_ylabel(ylabel)
    ax.set_title(f"{ylabel} convergence")
plt.show()
```

```
Epoch: 1/20, Step: 234/234, Loss: 3.8457, Acc: 0.2349
Epoch: 2/20, Step: 234/234, Loss: 1.8757, Acc: 0.4835
Epoch: 3/20, Step: 234/234, Loss: 1.3893, Acc: 0.5859
Epoch: 4/20, Step: 234/234, Loss: 1.1566, Acc: 0.6438
Epoch: 5/20, Step: 234/234, Loss: 1.0204, Acc: 0.6813
Epoch: 6/20, Step: 234/234, Loss: 0.9279, Acc: 0.7075
Epoch: 7/20, Step: 234/234, Loss: 0.8710, Acc: 0.7230
Epoch: 8/20, Step: 234/234, Loss: 0.8177, Acc: 0.7410
Epoch: 9/20, Step: 234/234, Loss: 0.7759, Acc: 0.7533
Epoch: 10/20, Step: 234/234, Loss: 0.7427, Acc: 0.7626
Epoch: 11/20, Step: 234/234, Loss: 0.7126, Acc: 0.7729
Epoch: 12/20, Step: 234/234, Loss: 0.6903, Acc: 0.7794
Epoch: 13/20, Step: 234/234, Loss: 0.6670, Acc: 0.7885
Epoch: 14/20, Step: 234/234, Loss: 0.6511, Acc: 0.7919
Epoch: 15/20, Step: 234/234, Loss: 0.6320, Acc: 0.7977
Epoch: 16/20, Step: 234/234, Loss: 0.6131, Acc: 0.8040
Epoch: 17/20, Step: 234/234, Loss: 0.6041, Acc: 0.8082
Epoch: 18/20, Step: 234/234, Loss: 0.5901, Acc: 0.8120
Epoch: 19/20, Step: 234/234, Loss: 0.5775, Acc: 0.8148
Epoch: 20/20, Step: 234/234, Loss: 0.5680, Acc: 0.8193
```



1.2.4 1.4 Evaluating the Model

Evaluate the model taking care that the statistics should not be used from the test set!

Explain why the evaluation needs to be treated differently.

Print the accuracy of both the train and test set in evaluation mode. Your accuracy should be close to ~80% on both the train and test sets.

[ ]:
```
### TODO: Evaluate the network
def evaluation(my_nn, param_dict, layers_stats_list, test_loader, device):
    total_test = 0
    total_correct = 0
    with torch.no_grad():
        for i, (data, label) in enumerate(test_loader):
            data = data.to(device, dtype=torch.float32)
            label = label.to(device, dtype=torch.long)
            output = my_nn(data, param_dict, layers_stats_list, train=False)
            total_correct += (output.argmax(dim=1) == label).sum().item()
            total_test += data.shape[0]
        return total_correct/total_test

train_acc = evaluation(my_nn, param_dict, layers_stats_list, train_loader,␣
 ↪device)
test_acc = evaluation(my_nn, param_dict, layers_stats_list, test_loader, device)
print(f"Train accuracy: {train_acc:.4f}, Test accuracy: {test_acc:.4f}")
```

```
Train accuracy: 0.8260, Test accuracy: 0.8322
```

1.3 Exercise 2: Modular Batch Normalization (OPTIONAL)

1.3.1 2.1 Batch Normalization Module (OPTIONAL)

Implement a `torch.nn.Module` that performs the batch normalization operation.

You will need to use the `register_buffer` in the `__init__` call of your custom `nn.Module` class to create variables that are not in the computation graph but tracked by `nn.Module`. Registering the buffer statistics for example allows the tensor to be moved onto the gpu when `model.cuda()` is called.

Hint: You can use the `.training` attribute of `torch.nn.Module` to detect if the model is in `.train()` mode or `.eval()` mode (example).

[ ]:
```python
import torch.nn as nn
import torch.nn.functional as F

class myBatchnorm(nn.Module):
    def __init__(self, num_features, epsilon=1e-3, momentum=.1):
        super(myBatchnorm,self).__init__()
        self.epsilon = epsilon
        self.m = momentum

        ### TODO: Initialize the `running_mean` and `running_var`
        ### register buffers with 0s and 1s respectively.
        self.register_buffer("running_mean", torch.zeros(num_features))
        self.register_buffer("running_var", torch.ones(num_features))


        ### TODO: Initialize the gamma and beta parameters
        self.gamma = nn.Parameter(torch.ones(num_features,requires_grad=True))
        self.beta = nn.Parameter(torch.zeros(num_features,requires_grad=True))


    def forward(self, x):
        ### TODO: perform batch normalization
        ### HINT: use nn.Module's .training attribute
        mean = self.running_mean
        var = self.running_var
        if self.training:
            curr_mean = x.mean(dim=0)
            curr_var = x.var(dim=0)
            mean = self.m*curr_mean + (1-self.m)*mean
            var = self.m*curr_var + (1-self.m)*var
            self.running_mean = mean
            self.running_var = var
            x = (x-curr_mean)/torch.sqrt(curr_var+self.epsilon)
            x = self.gamma*x + self.beta
            return x
        else:
            x = (x-mean)/torch.sqrt(var+self.epsilon)
            x = self.gamma*x + self.beta
            return x
```

```python
# Modify this class with your custom batchnorm
class Model(nn.Module):
    def __init__(self, h1_siz, h2_siz):
        super(Model, self).__init__()
        self.linear1 = nn.Linear(28*28, h1_siz)
        self.linear2 = nn.Linear(h1_siz, h2_siz)
        self.linear3 = nn.Linear(h2_siz, 10)

        ### TODO: initialize batch normalization layers
        self.bn1 = myBatchnorm(h1_siz)
        self.bn2 = myBatchnorm(h2_siz)

        self.init_weights()

    def init_weights(self):
        self.linear1.weight.data.uniform_(-1, 1)
        self.linear1.bias.data.uniform_(-1, 1)
        self.linear2.weight.data.uniform_(-1, 1)
        self.linear2.bias.data.uniform_(-1, 1)
        self.linear3.weight.data.uniform_(-1, 1)
        self.linear3.bias.data.uniform_(-1, 1)


    def forward(self, x):
        x = x.view(-1, 28*28)
        x = self.linear1(x)

        ### TODO: add batch normalization layer
        x = self.bn1(x)

        ###
        x = self.linear2(F.relu(x))
        ### TODO: add batch normalization layer
        x = self.bn2(x)

        ###
        x = F.relu(x)
        return self.linear3(x)
```

1.3.2 2.2 Training the Model (OPTIONAL)

Repeat training and overlay the training curves to those from (1.3-1.4) and validate it achieves similar test acc. In order to achieve the same behavior as your `train=False/train=True`, you will need to use `.eval()` and `.train()` methods on your model.

You should get roughly close values to the following:

[ ]:
```python
def train(model, optimizer, train_loader, history_frequency=50):
    r"""Iterates over train_loader and optimizes model using pre-initialized
    optimizer.

    Args:
        model (torch.nn.Module): Model to be trained
        optimizer (torch.optim.Optimizer): initialized optimizer with lr and
            model parameters
        train_loader (torch.utils.data.DataLoader): Training set data loader
        history_frequency (int): Frequency for the minibatch metrics to be
            stored in minibatch_losses and minibatch_accuracies

    Returns:
        minibatch_losses (list of float): Minibatch loss every over the
            training progress
        minibatch_accuracies (list of float): Minibatch accuracy over the
            training progress
    """
    minibatch_losses = []
    minibatch_accuracies = []

    ### TODO: Use `.train()` to put model in training state
    model.train()

    total_train = 0
    total_correct = 0
    total_loss = 0

    for i,(data,label) in enumerate(train_loader):
        ### TODO: perform forward pass and backpropagation
        ### TODO: store the loss and accuracy in `minibatch_losses` and
        ### `minibatch_accuracies` every `history_frequency`th iteration
        data = data.to(device, dtype=torch.float32)
        label = label.to(device, dtype=torch.long)
        output = model(data)
        loss = torch.nn.functional.cross_entropy(output, label)

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

        m_batch = data.shape[0]
        total_train += m_batch
        total_loss += loss.item()*m_batch
        total_correct += (output.argmax(dim=1) == label).sum().item()
```

```python
        if i % history_frequency == 0:
            minibatch_losses.append(total_loss/total_train)
            minibatch_accuracies.append(total_correct/total_train)
            total_train = 0
            total_correct = 0
            total_loss = 0


    return minibatch_losses, minibatch_accuracies

def test(model, test_loader):
    r"""Iterate over test_loader to compute the accuracy of the trained model

    Args:
        model (torch.nn.Module): Model to be evaluated
        test_loader (torch.utils.data.DataLoader): Testing set data loader

    Returns:
        accuracy (float): Model accuracy on test set
        loss (float): Model loss on test set
    """
    accuracy = 0
    loss = 0

    ### TODO: Use `.eval()` to put model in evaluation state
    model.eval()
    total_correct = 0
    total_train = 0

    with torch.no_grad():
        for i,(data,label) in enumerate(test_loader):
            ### TODO: perform forward pass and compute the loss and accuracy
            data = data.to(device, dtype=torch.float32)
            label = label.to(device, dtype=torch.long)
            output = model(data)
            loss += torch.nn.functional.cross_entropy(output,␣
↪label,reduction='sum').item()
            total_correct += (output.argmax(dim=1) == label).sum().item()
            total_train += data.shape[0]

        accuracy = total_correct/total_train
        loss = loss/total_train
    return (loss, accuracy)
```

[ ]:
```python
# training hyper_parameters
lr = 0.01
```

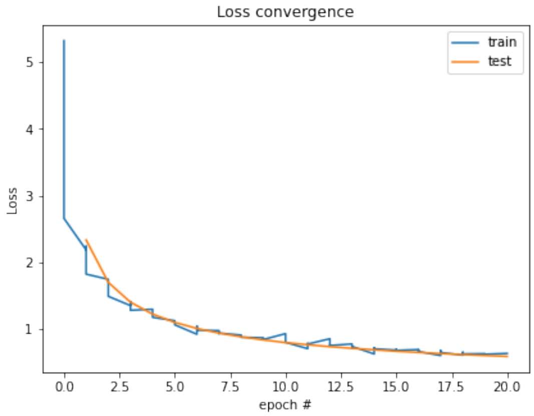
```python
num_epochs = 20

### TODO: initialize the model and the optimizer
### Reminder: The running_mean and running_variance are not updated by gradient␣
↪descent!
model = Model(h1_size, h2_size).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=lr)

train_losses = []
train_accuracies = []

test_losses = []
test_accuracies = []

for epoch in range(num_epochs):
    train_loss, train_accuracy = train(model, optimizer, train_loader)
    train_losses.extend(train_loss)
    train_accuracies.extend(train_accuracy)
    test_loss, test_accuracy = test(model, test_loader)
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)

    print('Epoch: {}, Train Loss: {:.4f}, Train Accuracy: {:.4f}, Test Loss: {:.
↪4f}, Test Accuracy: {:.4f}'.format(epoch+1, train_loss[-1],␣
↪train_accuracy[-1], test_loss, test_accuracy))
```

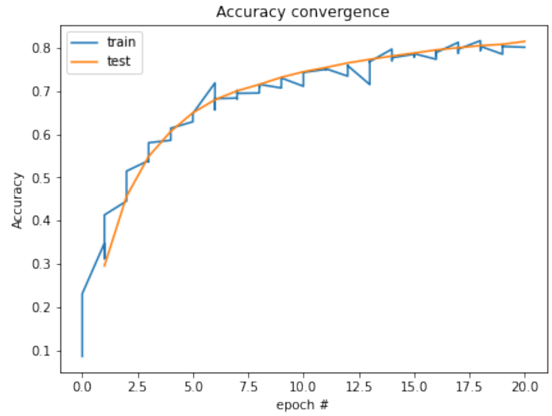
```
Epoch: 1, Train Loss: 2.6598, Train Accuracy: 0.2311, Test Loss: 2.3363, Test
Accuracy: 0.2955
Epoch: 2, Train Loss: 1.8239, Train Accuracy: 0.4132, Test Loss: 1.7001, Test
Accuracy: 0.4561
Epoch: 3, Train Loss: 1.4914, Train Accuracy: 0.5146, Test Loss: 1.4033, Test
Accuracy: 0.5486
Epoch: 4, Train Loss: 1.2817, Train Accuracy: 0.5803, Test Loss: 1.2248, Test
Accuracy: 0.6066
Epoch: 5, Train Loss: 1.1767, Train Accuracy: 0.6138, Test Loss: 1.1001, Test
Accuracy: 0.6496
Epoch: 6, Train Loss: 1.0646, Train Accuracy: 0.6480, Test Loss: 1.0097, Test
Accuracy: 0.6798
Epoch: 7, Train Loss: 0.9837, Train Accuracy: 0.6823, Test Loss: 0.9398, Test
Accuracy: 0.7003
Epoch: 8, Train Loss: 0.9400, Train Accuracy: 0.6946, Test Loss: 0.8839, Test
Accuracy: 0.7150
Epoch: 9, Train Loss: 0.8773, Train Accuracy: 0.7153, Test Loss: 0.8382, Test
Accuracy: 0.7317
Epoch: 10, Train Loss: 0.8441, Train Accuracy: 0.7302, Test Loss: 0.7976, Test
Accuracy: 0.7445
Epoch: 11, Train Loss: 0.8008, Train Accuracy: 0.7430, Test Loss: 0.7662, Test
Accuracy: 0.7542
```

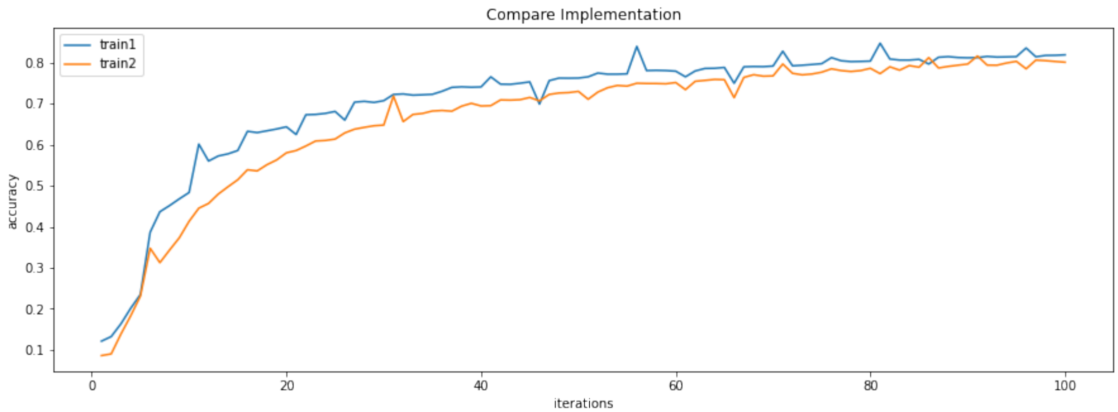
```
Epoch: 12, Train Loss: 0.7756, Train Accuracy: 0.7517, Test Loss: 0.7364, Test
Accuracy: 0.7648
Epoch: 13, Train Loss: 0.7535, Train Accuracy: 0.7587, Test Loss: 0.7118, Test
Accuracy: 0.7731
Epoch: 14, Train Loss: 0.7405, Train Accuracy: 0.7679, Test Loss: 0.6897, Test
Accuracy: 0.7809
Epoch: 15, Train Loss: 0.7042, Train Accuracy: 0.7770, Test Loss: 0.6681, Test
Accuracy: 0.7877
Epoch: 16, Train Loss: 0.6813, Train Accuracy: 0.7864, Test Loss: 0.6503, Test
Accuracy: 0.7951
Epoch: 17, Train Loss: 0.6638, Train Accuracy: 0.7890, Test Loss: 0.6332, Test
Accuracy: 0.8001
Epoch: 18, Train Loss: 0.6496, Train Accuracy: 0.7968, Test Loss: 0.6178, Test
Accuracy: 0.8050
Epoch: 19, Train Loss: 0.6282, Train Accuracy: 0.8034, Test Loss: 0.6043, Test
Accuracy: 0.8080
Epoch: 20, Train Loss: 0.6352, Train Accuracy: 0.8013, Test Loss: 0.5908, Test
Accuracy: 0.8147
```

[ ]:

```
### TODO: Visualize training curves
train_acc_my_batchnorm = train_accuracies
fig, ax = plt.subplots(1,2, figsize=(15,5))

for ax, (data, train_data, test_data) in zip(ax,
 ↪[("loss",train_losses,test_losses),("accuracy",train_accuracies,test_accuracies)]):
 ↪
    ax.plot(np.linspace(0,len(test_data), len(train_data)).astype(int),
 ↪train_data, label="train")
    ax.plot(np.arange(len(test_data))+1, test_data, label="test")
    ax.set_xlabel("epoch #")
    ax.legend()
    ylabel = "Loss" if data == "loss" else "Accuracy"
    ax.set_ylabel(ylabel)
    ax.set_title( ylabel + " convergence")
fig, ax = plt.subplots(1,1, figsize=(15,5))
acc_history = train_track["acc"]
ax.plot(np.arange(len(acc_history))+1, acc_history, label="train1")
ax.plot(np.arange(len(train_accuracies))+1, train_accuracies, label="train2")
ax.set_xlabel("iterations")
ax.set_ylabel("accuracy")
ax.legend()
ax.set_title("Compare Implementation")
plt.show()
```







1.3.3 2.3 PyTorch's nn.BatchNorm1d (OPTIONAL)

Finally repeat all these steps using PyTorch's `nn.BatchNorm1d` module and validate that the training curves match those from (1.3-1.4) and (2.2)

[ ]:

```
import torch.nn as nn
import torch.nn.functional as F

# Modify this class with your custom batchnorm
class Model(nn.Module):
    def __init__(self, h1_siz, h2_siz):
        super(Model, self).__init__()
        self.linear1 = nn.Linear(28*28, h1_siz)
        self.linear2 = nn.Linear(h1_siz, h2_siz)
        self.linear3 = nn.Linear(h2_siz, 10)
        ### TODO: add batch normalization module
        self.batchnorm1 = nn.BatchNorm1d(h1_siz)
        self.batchnorm2 = nn.BatchNorm1d(h2_siz)
```

```
        self.init_weights()

    def init_weights(self):
        self.linear1.weight.data.uniform_(-1,1)
        self.linear1.bias.data.uniform_(-1,1)
        self.linear2.weight.data.uniform_(-1,1)
        self.linear2.bias.data.uniform_(-1,1)
        self.linear3.weight.data.uniform_(-1,1)
        self.linear3.bias.data.uniform_(-1,1)

        self.batchnorm1.weight.data.fill_(1)
        self.batchnorm1.bias.data.zero_()
        self.batchnorm2.weight.data.fill_(1)
        self.batchnorm2.bias.data.zero_()


    def forward(self, x):
        x = x.view(-1, 28*28)
        x = x.view(-1, 28*28)
        x = self.linear1(x)

        ### TODO: add batch normalization layer
        x = self.batchnorm1(x)

        x = F.relu(x)

        ###
        x = self.linear2(x)
        ### TODO: add batch normalization layer
        x = self.batchnorm2(x)
        ###
        x = F.relu(x)
        return self.linear3(x)
```

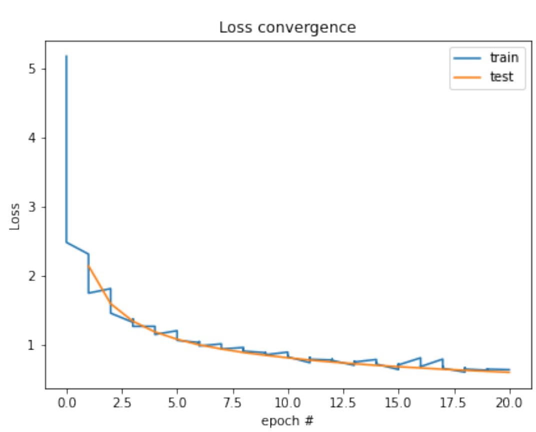
[ ]:
```
# training hyper_parameters
lr = 0.01
num_epochs = 20

### TODO: initialize the model and the optimizer
model = Model(h1_size, h2_size).to(device)
optimizer = torch.optim.SGD(model.parameters(), lr=lr)

train_losses = []
train_accuracies = []
```

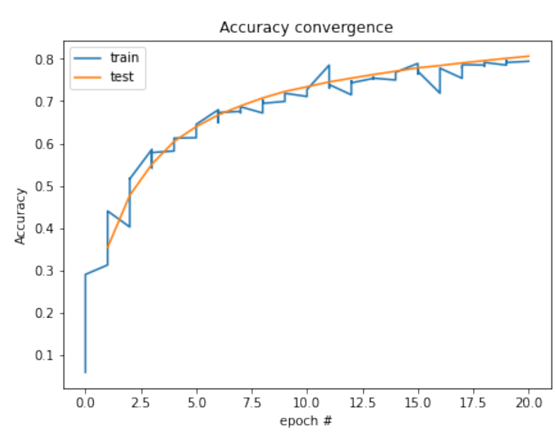
```python
test_losses = []
test_accuracies = []

for epoch in range(num_epochs):
    train_loss, train_accuracy = train(model, optimizer, train_loader)
    train_losses.extend(train_loss)
    train_accuracies.extend(train_accuracy)
    test_loss, test_accuracy = test(model, test_loader)
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)

    print('Epoch: {}, Train Loss: {:.4f}, Train Accuracy: {:.4f}, Test Loss: {:.
↪4f}, Test Accuracy: {:.4f}'.format(epoch+1, train_loss[-1],␣
↪train_accuracy[-1], test_loss, test_accuracy))
```

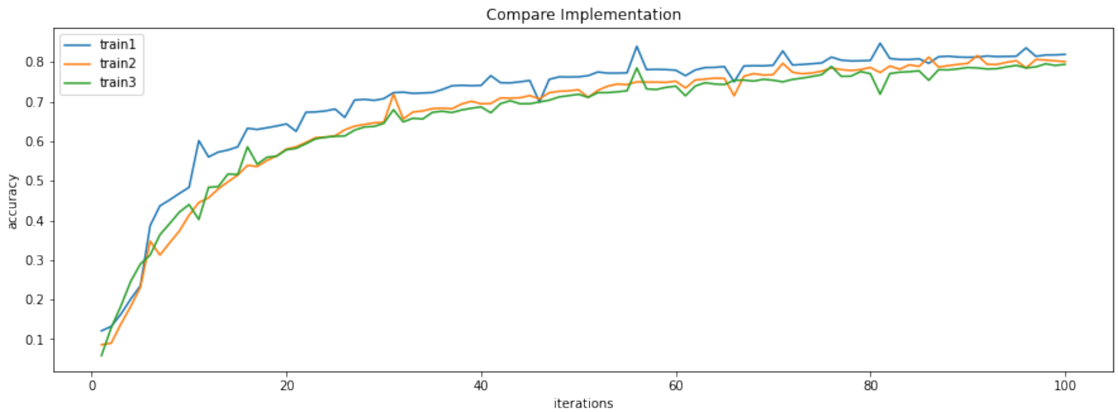
```
Epoch: 1, Train Loss: 2.4835, Train Accuracy: 0.2900, Test Loss: 2.1441, Test
Accuracy: 0.3539
Epoch: 2, Train Loss: 1.7508, Train Accuracy: 0.4401, Test Loss: 1.5978, Test
Accuracy: 0.4771
Epoch: 3, Train Loss: 1.4619, Train Accuracy: 0.5159, Test Loss: 1.3456, Test
Accuracy: 0.5509
Epoch: 4, Train Loss: 1.2719, Train Accuracy: 0.5784, Test Loss: 1.1915, Test
Accuracy: 0.6047
Epoch: 5, Train Loss: 1.1545, Train Accuracy: 0.6124, Test Loss: 1.0852, Test
Accuracy: 0.6396
Epoch: 6, Train Loss: 1.0696, Train Accuracy: 0.6450, Test Loss: 1.0048, Test
Accuracy: 0.6675
Epoch: 7, Train Loss: 0.9893, Train Accuracy: 0.6730, Test Loss: 0.9453, Test
Accuracy: 0.6885
Epoch: 8, Train Loss: 0.9442, Train Accuracy: 0.6866, Test Loss: 0.8947, Test
Accuracy: 0.7074
Epoch: 9, Train Loss: 0.9142, Train Accuracy: 0.6948, Test Loss: 0.8546, Test
Accuracy: 0.7228
Epoch: 10, Train Loss: 0.8620, Train Accuracy: 0.7186, Test Loss: 0.8178, Test
Accuracy: 0.7340
Epoch: 11, Train Loss: 0.8312, Train Accuracy: 0.7273, Test Loss: 0.7854, Test
Accuracy: 0.7453
Epoch: 12, Train Loss: 0.7963, Train Accuracy: 0.7391, Test Loss: 0.7567, Test
Accuracy: 0.7542
Epoch: 13, Train Loss: 0.7796, Train Accuracy: 0.7432, Test Loss: 0.7323, Test
Accuracy: 0.7628
Epoch: 14, Train Loss: 0.7567, Train Accuracy: 0.7541, Test Loss: 0.7091, Test
Accuracy: 0.7706
Epoch: 15, Train Loss: 0.7277, Train Accuracy: 0.7681, Test Loss: 0.6889, Test
Accuracy: 0.7787
Epoch: 16, Train Loss: 0.7150, Train Accuracy: 0.7707, Test Loss: 0.6712, Test
Accuracy: 0.7840
Epoch: 17, Train Loss: 0.6903, Train Accuracy: 0.7778, Test Loss: 0.6531, Test
```

```
Accuracy: 0.7904
Epoch: 18, Train Loss: 0.6666, Train Accuracy: 0.7863, Test Loss: 0.6373, Test
Accuracy: 0.7957
Epoch: 19, Train Loss: 0.6545, Train Accuracy: 0.7916, Test Loss: 0.6233, Test
Accuracy: 0.8012
Epoch: 20, Train Loss: 0.6458, Train Accuracy: 0.7941, Test Loss: 0.6082, Test
Accuracy: 0.8062
```

[ ]:
```
### TODO: Visualize training curves

fig, ax = plt.subplots(1,2, figsize=(15,5))

for ax, (data, train_data, test_data) in zip(ax,␣
 ↪[("loss",train_losses,test_losses),("accuracy",train_accuracies,test_accuracies)]):
 ↪
    ax.plot(np.linspace(0,len(test_data), len(train_data)).astype(int),␣
 ↪train_data, label="train")
    ax.plot(np.arange(len(test_data))+1, test_data, label="test")
    ax.set_xlabel("epoch #")
    ax.legend()
    ylabel = "Loss" if data == "loss" else "Accuracy"
    ax.set_ylabel(ylabel)
    ax.set_title( ylabel + " convergence")

fig, ax = plt.subplots(1,1, figsize=(15,5))
acc_history = train_track["acc"]
ax.plot(np.arange(len(acc_history))+1, acc_history, label="train1")
ax.plot(np.arange(len(train_acc_my_batchnorm))+1, train_acc_my_batchnorm,␣
 ↪label="train2")
ax.plot(np.arange(len(train_accuracies))+1, train_accuracies, label="train3")
ax.set_xlabel("iterations")
ax.set_ylabel("accuracy")
ax.legend()
ax.set_title("Compare Implementation")
plt.show()
```







Compare Implementation
train1
train2
train3
accuracy
iterations
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
20
40
60
80
100