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.

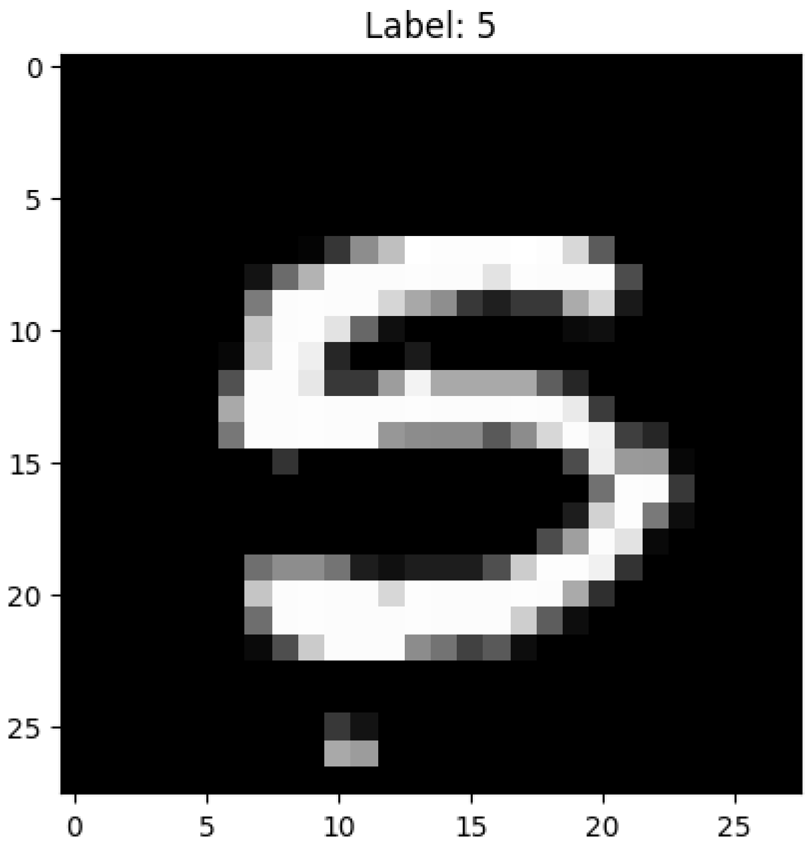
0 Initialization

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

In [1]:
```
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)
```

```
/home/ziruiqiu/anaconda3/envs/DL2/lib/python3.9/site-packages/tqdm/au
to.py:22: TqdmWarning: IProgress not found. Please update jupyter and
ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_inst
all.html (https://ipywidgets.readthedocs.io/en/stable/user_install.ht
ml)
  from .autonotebook import tqdm as notebook_tqdm
```

In [2]:
```
# 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');
```



Exercise 1: Functional Batch Normalization

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 (https://moodle.concordia.ca/moodle/pluginfile.php/5877105/mod_resource/content/2/Lectu](https://moodle.concordia.ca/moodle/pluginfile.php/5877105/mod_resource/content/2/Lectu) 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=<VarBackward1>)
```

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

In [20]:
```
device = "cpu"
# 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.zeros(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.float
gamma = torch.ones(num_features, requires_grad=True, dtype=torch.float

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

    Args:
        inputs (torch.tensor): Batch of inputs of shape (N, F), where
            the minibatch size, and F is the number of features.
        beta (torch.tensor): Batch normalization beta variable of shap
        gamma (torch.tensor): Batch normalization gamma variable of sh
        stats_dict (dict of torch.tensor): Dictionary containing the r
            mean and variance. Expects dictionary to contain keys 'rur
            and 'running_var', with values being `torch.tensor`s of sh
        train (bool): Determines whether batch norm is in train mode c
            Default: True
        eps (float): Constant for numeric stability.
        momentum (float): The momentum value for updating the running
            variance during training.

    Returns:
        torch.tensor: Batch normalized inputs, of shape (N, F)
    """
    ### TODO: Fill out this function
    minibatch_mean = stats_dict["running_mean"]
    minibatch_var = stats_dict["running_var"]

    if train:
        # compute batch statistics
        batch_mean = inputs.mean(0)
        batch_var = inputs.var(0)

        # update population statistics
        minibatch_mean = (minibatch_mean * (1 - momentum)) + (batch_me
```

```
        minibatch_var = (minibatch_var * (1 - momentum)) + (batch_var
        stats_dict["running_mean"] = minibatch_mean
        stats_dict["running_var"] = minibatch_var

        # standardize inputs
        return gamma * (inputs - batch_mean) / ((batch_var + eps) ** 0
    else:
        # inference mode: standardize with train statistics
        return gamma * (inputs - minibatch_mean) / ((minibatch_var + e

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

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

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

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!

In [21]:

```
# Initialize model hiden layer sizes
h1_size = 50
h2_size = 50

### TODO: Initialize the beta and gamma parameters
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]

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), wher
            the number of input samples, and H and W are the image hei
            width respectively.
        param_dict (dict of torch.tensor): Dictionary containing the p
            of the neural network. Expects dictionary keys to be of fo
            "W#", "b#", "beta#" and "gamma#" where # is the layer numb
        layers_stats_list (list of dict of torch.tensor): List of dict
            containing running means and variances for each layer. Lis
            is equal to the number of hidden layers.
        train (bool): Determines whether batch norm is in train mode o
```

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

    ### TODO: use your complete batchnorm function
    x = batchnorm(x, param_dict['beta0'], param_dict['gamma0'], layers
    x = torch.relu(x)

    # 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 p
            of the neural network. Expects dictionary keys to be of fo
            "W#", "b#", "beta#" and "gamma#" where # is the layer numb
        layers_stats_list (list of dict of torch.tensor): List of dict
            containing running means and variances for each layer. Lis
            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 stat
    in `layers_stats_list`.

    Args:
        param_dict (dict of torch.tensor): Dictionary containing the p
            of the neural network. Expects dictionary keys to be of fo
            "W#", "b#", "beta#" and "gamma#" where # is the layer numb
        layers_stats_list (list of dict of torch.tensor): List of dict
            containing running means and variances for each layer. Lis
            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_dict in layers_stats_list:
        layer_stats_dict["running_mean"] = torch.zeros_like(layer_stat
        layer_stats_dict["running_var"] = torch.ones_like(layer_stats_
```

1.3 Training the Model

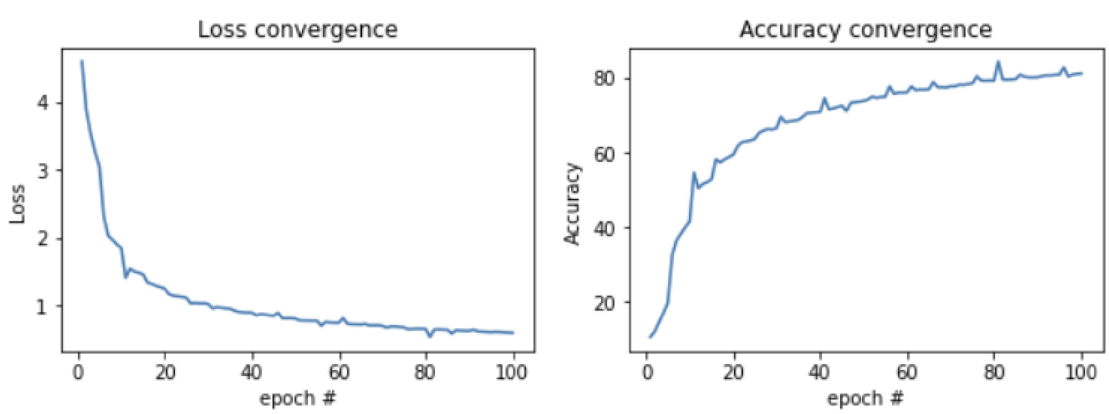
Train the model on the MNIST dataset with 20 epochs and `lr=0.01` with SGD and without momentum (as per lab 2). Since you are dealing with the MNIST dataset and you are required to perform multiclass classification, you will use the cross entropy loss during training (similar to lab2).

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:

```
Epoch 01: train loss 3.0513, train acc 19.77%
Epoch 02: train loss 1.8476, train acc 41.54%
Epoch 03: train loss 1.4571, train acc 52.74%
Epoch 04: train loss 1.2469, train acc 59.46%
Epoch 05: train loss 1.1162, train acc 63.60%
```

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

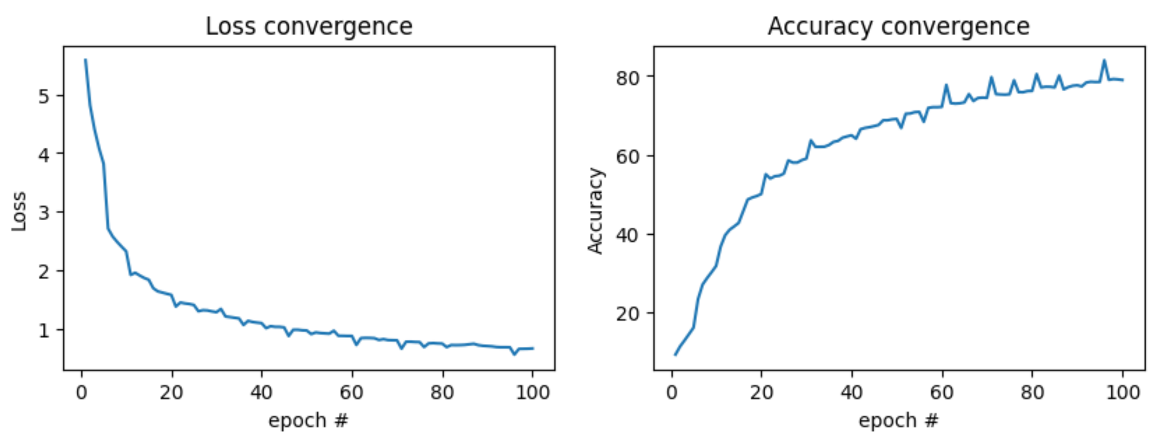


In [42]:
```
from torch.optim import SGD

# 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)
train_record = {
    'train_loss_list': [],
    'train_acc_list': [],
}
for epoch in range(num_epochs):
    train_size_sum = 0
    train_loss_sum = 0
    train_correct_sum = 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
        scores = my_nn(data, param_dict, layers_stats_list, train=True
        loss = torch.nn.functional.cross_entropy(scores, label)

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

        # loss and accuracy
        minibatch_size = len(data)
        train_size_sum += minibatch_size
        train_loss_sum += (minibatch_size * loss).item()
        train_correct_sum += (scores.max(dim=1)[1] == label).sum()
        if i%50 == 0:
            train_loss = train_loss_sum / train_size_sum
            train_acc = 100 * (float(train_correct_sum) / train_size_s
            train_record['train_loss_list'].append(train_loss)
            train_record['train_acc_list'].append(train_acc)

    # print training progress
    print(f'Epoch {epoch+1:02d}: train loss {train_loss:.4f}, train ac
```

```
Epoch 01: train loss 3.8192, train acc 16.21%
Epoch 02: train loss 2.3233, train acc 31.85%
Epoch 03: train loss 1.8382, train acc 42.75%
Epoch 04: train loss 1.5797, train acc 50.07%
Epoch 05: train loss 1.4078, train acc 55.24%
Epoch 06: train loss 1.2842, train acc 59.05%
Epoch 07: train loss 1.1785, train acc 62.48%
Epoch 08: train loss 1.0975, train acc 64.97%
Epoch 09: train loss 1.0248, train acc 67.28%
Epoch 10: train loss 0.9716, train acc 69.10%
Epoch 11: train loss 0.9186, train acc 70.89%
Epoch 12: train loss 0.8792, train acc 72.15%
Epoch 13: train loss 0.8403, train acc 73.29%
Epoch 14: train loss 0.8042, train acc 74.45%
Epoch 15: train loss 0.7743, train acc 75.35%
Epoch 16: train loss 0.7485, train acc 76.20%
Epoch 17: train loss 0.7272, train acc 77.09%
Epoch 18: train loss 0.7066, train acc 77.63%
Epoch 19: train loss 0.6851, train acc 78.47%
Epoch 20: train loss 0.6655, train acc 78.98%
```

In [78]:

```
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
for ax, metric in zip(axes, ['loss', 'acc']):
    loss_history = train_record[f'train_{metric}_list']
    ax.plot(np.arange(len(loss_history))+1, loss_history)
    ax.set_xlabel('epoch #')
    name = "Loss" if metric == "loss" else "Accuracy"
    ax.set_ylabel(name)
    ax.title.set_text(f'{name} convergence')
plt.show()
```



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.

In [79]:
```
### TODO: Evaluate the network
def evaluate(my_nn_func, param_dict, layers_stats_list, loader, device
    """Returns the accuracy of the model in evaluation mode on loader

    Args:
        my_nn_func (function): Function for performing the forward pas
            of the neural network.
        param_dict (dict of torch.tensor): Dictionary containing the p
            of the neural network. Expects dictionary keys to be of fo
            "W#", "b#", "beta#" and "gamma#" where # is the layer numb
        layers_stats_list (list of dict of torch.tensor): List of dict
            containing running means and variances for each layer. Lis
            is equal to the number of hidden layers.
        loader (torch.utils.data.DataLoader): DataLoader object

    Returns:
        (float): Accuracy percentage
    """
    num_correct = 0
    num_total = 0
    with torch.no_grad():
        for i, (data, label) in enumerate(train_loader):

            data = data.to(device, dtype=torch.float32)
            label = label.to(device, dtype=torch.long)

            # forward pass
            scores = my_nn_func(data, param_dict, layers_stats_list, t

            # Compute predictions
            preds = scores.max(dim=1)[1]
            num_correct += (preds == label).sum()
            num_total += len(preds)

        # compute accuracy
        acc = 100 * (float(num_correct) / num_total)

    return acc

train_acc = evaluate(my_nn, param_dict, layers_stats_list, train_loade
test_acc = evaluate(my_nn, param_dict, layers_stats_list, test_loader,
print(f'accuracy: train {train_acc:.2f}%, test {test_acc:.2f}%')
```

```
accuracy: train 79.36%, test 79.33%
```

Exercise 2: Modular Batch Normalization (OPTIONAL)

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 (https://github.com/pytorch/pytorch/blob/fcf8b712348f21634044a5d76a69a59727756357/torch/

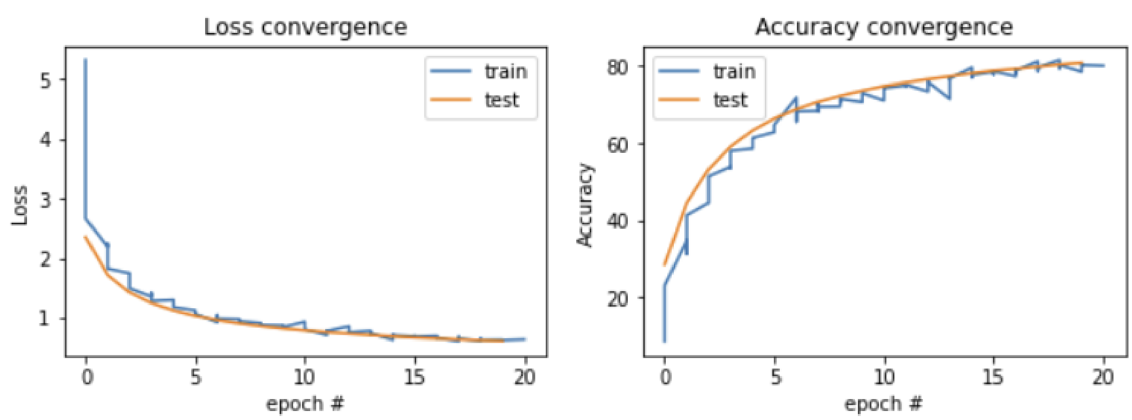
In [80]:
```
import torch.nn as nn
import torch.nn.functional as F

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_gr
        self.beta = nn.Parameter(torch.zeros(num_features, requires_gr
    def forward(self, x):
        ### TODO: perform batch normalization
        ### HINT: use nn.Module's .training attribute
        pop_mean = self.running_mean
        pop_var = self.running_var

        if self.training:
            # compute batch statistics
            batch_mean = x.mean(0)
            batch_var = x.var(0)

            # update population statistics
            pop_mean = (pop_mean * (1 - self.m)) + (batch_mean * self.
            pop_var = (pop_var * (1 - self.m)) + (batch_var * self.m)
            self.running_mean = pop_mean
            self.running_var = pop_var

            # standardize x
            return self.gamma * (x - batch_mean) / ((batch_var + self.
        else:
            # inference mode: standardize with train statistics
            return self.gamma * (x - pop_mean) / ((pop_var + self.epsi

# 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_size, 10)
        ### TODO: initialize batch normalization layers
        self.bn1 = myBatchnorm(h1_size)
        self.bn2 = myBatchnorm(h2_size)
        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)
```

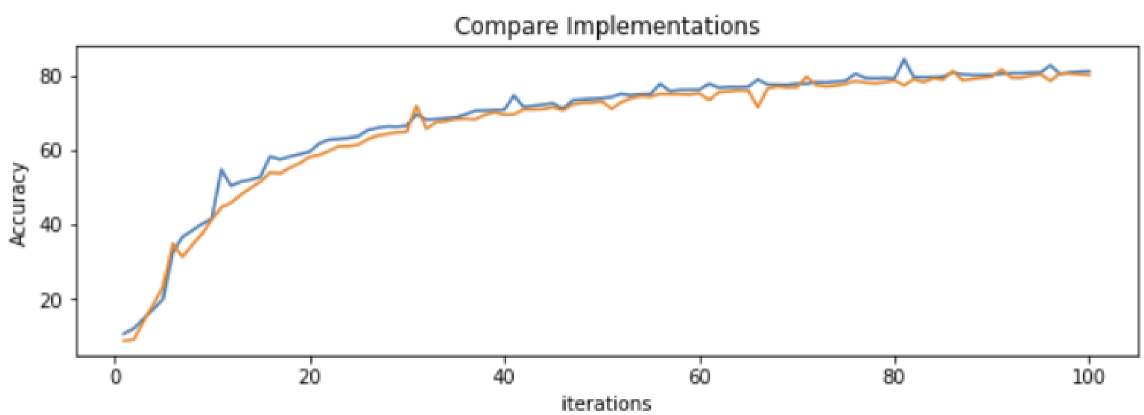
```
        self.linear3.bias.data.uniform_(-1,1)
    def forward(self, x):
        x = x.view(-1, 28*28)
        x = self.linear1(x)

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

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

        x = torch.relu(x)

        return self.linear3(x)
```

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:





In [81]:
```
def train(model, optimizer, train_loader, history_frequency=50):
    r"""Iterates over train_loader and optimizes model using pre-initi
    optimizer.

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

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

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

    total_train_size = 0
    total_train_loss = 0
    total_train_correct = 0
    for i,(data,label) in enumerate(train_loader):
        ### TODO: perform forward pass and backpropagation
        ### TODO: store the loss and accuracy in `minibatch_losses` an
        ### `minibatch_accuracies` every `history_frequency`th iterati
        # move inputs to desired device and dtype
        data = data.to(device, dtype=torch.float32)
        label = label.to(device, dtype=torch.long)

        # forward pass
        scores = model(data)
        loss = torch.nn.functional.cross_entropy(scores, label)

        # backward pass
        optimizer.zero_grad() # Zero out the gradients
        loss.backward() # Compute gradient of loss w.r.t. model parame
        optimizer.step() # Make a gradient update step

        # Epoch loss and accuracy average
        minibatch_size = len(data)
        total_train_size += minibatch_size
        total_train_loss += (minibatch_size * loss).item()
        total_train_correct += (scores.max(dim=1)[1] == label).sum()

        # update metrics
        if i % history_frequency == 0:
            # compute accuracy and average of loss every history_frequ
            train_history_loss = total_train_loss / total_train_size
            train_history_acc = 100 * (float(total_train_correct) / to
            minibatch_losses.append(train_history_loss)
            minibatch_accuracies.append(train_history_acc)
```

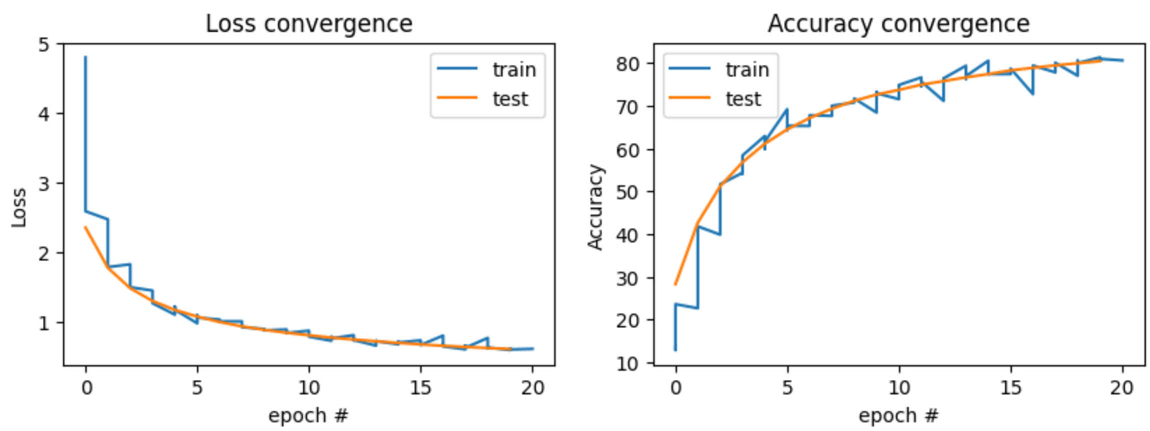
```
            total_train_size = 0
            total_train_loss = 0
            total_train_correct = 0

    return minibatch_losses, minibatch_accuracies

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

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

    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()
    num_correct = 0
    num_total = 0
    with torch.no_grad():  # temporarily set all requires_grad flags t
        for i, (data, label) in enumerate(train_loader):
            ### TODO: perform forward pass and compute the loss and ac

            # move inputs to desired device and dtype
            data = data.to(device, dtype=torch.float32)
            label = label.to(device, dtype=torch.long)

            # forward pass
            scores = model(data)

            # compute loss and number of accurate predictions
            loss += torch.nn.functional.cross_entropy(scores, label, r
            preds = scores.max(dim=1)[1]
            num_correct += (preds == label).sum().item()
            num_total += len(preds)

        # compute accuracy percentage
        accuracy = 100 * (float(num_correct) / num_total)
        loss /= num_total

    return (loss, accuracy)
```

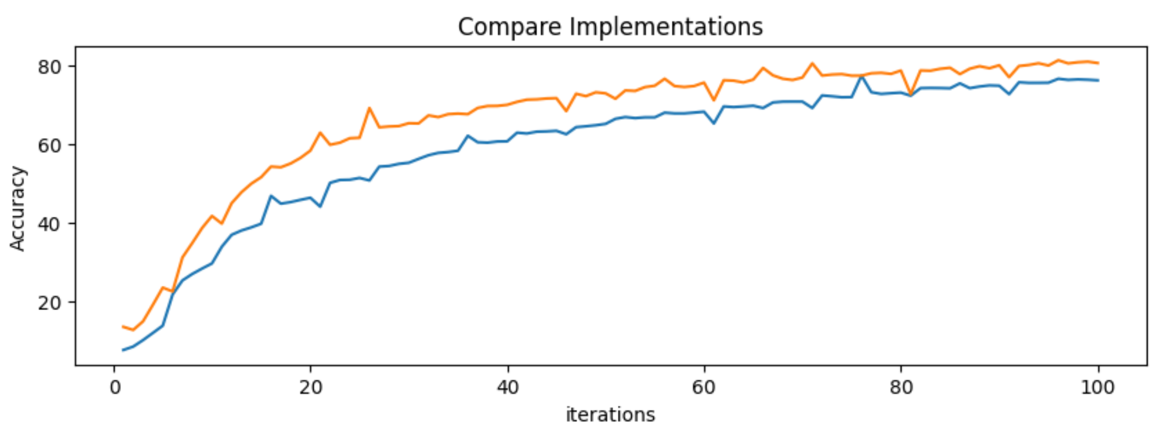
In [60]:

```
# training hyper_parameters
lr = 0.01
num_epochs = 20

### TODO: initialize the model and the optimizer
### Reminder: The running_mean and running_variance are not updated by
model = Model(h1_size, h2_size).to(device)
optimizer = torch.optim.SGD(model.parameters(),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 training progress
    print(
        f'Epoch {epoch+1:02d}: '
        f'train loss {train_loss[-1]:.4f}, train acc {train_accuracy[-
        f'test loss {test_loss:.4f}, test acc {test_accuracy:.2f}%'
        )
```

```
Epoch 01: train loss 2.4757, train acc 28.15%, test loss 2.2480, test
acc 31.99%
Epoch 02: train loss 1.8117, train acc 41.27%, test loss 1.7192, test
acc 44.24%
Epoch 03: train loss 1.5121, train acc 50.67%, test loss 1.4391, test
acc 52.43%
Epoch 04: train loss 1.3190, train acc 56.38%, test loss 1.2613, test
acc 58.11%
Epoch 05: train loss 1.1637, train acc 62.10%, test loss 1.1360, test
acc 62.57%
Epoch 06: train loss 1.0803, train acc 64.37%, test loss 1.0430, test
acc 65.84%
Epoch 07: train loss 1.0095, train acc 67.04%, test loss 0.9695, test
acc 68.35%
Epoch 08: train loss 0.9344, train acc 69.76%, test loss 0.9127, test
acc 70.39%
Epoch 09: train loss 0.9029, train acc 70.88%, test loss 0.8630, test
acc 72.12%
Epoch 10: train loss 0.8526, train acc 72.71%, test loss 0.8233, test
acc 73.55%
Epoch 11: train loss 0.8033, train acc 74.78%, test loss 0.7882, test
acc 74.69%
Epoch 12: train loss 0.7852, train acc 75.03%, test loss 0.7587, test
acc 75.69%
Epoch 13: train loss 0.7639, train acc 75.11%, test loss 0.7323, test
acc 76.57%
Epoch 14: train loss 0.7215, train acc 76.97%, test loss 0.7095, test
acc 77.28%
Epoch 15: train loss 0.7016, train acc 77.49%, test loss 0.6903, test
acc 77.97%
Epoch 16: train loss 0.6818, train acc 78.02%, test loss 0.6714, test
acc 78.58%
Epoch 17: train loss 0.6588, train acc 79.29%, test loss 0.6540, test
acc 79.15%
Epoch 18: train loss 0.6480, train acc 79.13%, test loss 0.6392, test
acc 79.62%
Epoch 19: train loss 0.6446, train acc 79.85%, test loss 0.6246, test
acc 80.10%
Epoch 20: train loss 0.6179, train acc 80.29%, test loss 0.6133, test
acc 80.50%
```

In [82]:
```
### TODO: Visualize training curves
fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for ax, (metric, train_metric, test_metric) in zip(axes, [("loss", tra
    ax.plot(np.linspace(0, len(test_metric), len(train_metric)).astype
    ax.plot(np.arange(len(test_metric)), test_metric, label="test")
    ax.set_xlabel('epoch #')
    ax.legend()
    name = "Loss" if metric == "loss" else "Accuracy"
    ax.set_ylabel(name)
    ax.title.set_text(f'{name} convergence')

fig, axes = plt.subplots(1, 1, figsize=(10, 3))
loss_history = train_history[f'train_acc_hist']
axes.plot(np.arange(len(loss_history))+1, loss_history, label='train_v
axes.plot(np.arange(len(train_accuracies))+1, train_accuracies, label=
axes.set_xlabel('iterations')
axes.set_ylabel(name)
axes.title.set_text('Compare Implementations')
plt.show()
```





2.3 PyTorch's nn.BatchNorm1d (OPTIONAL)

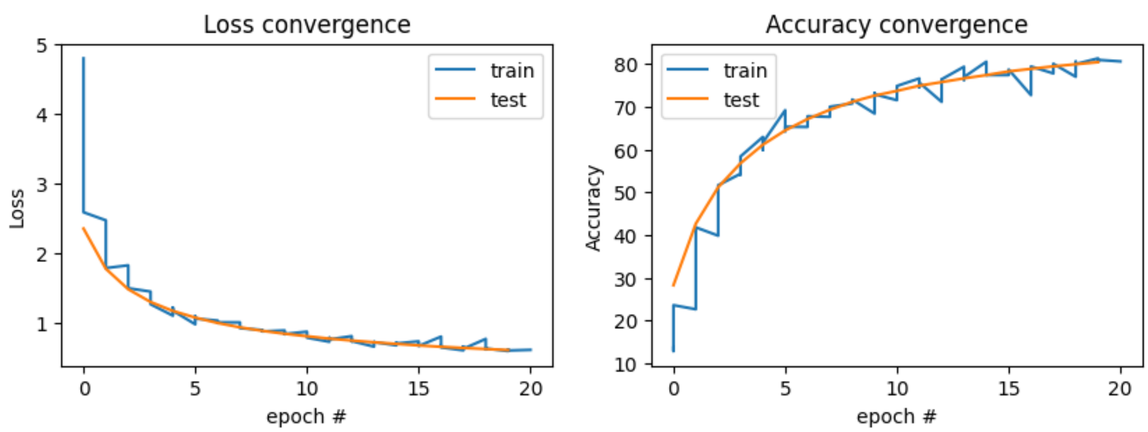
Finally repeat all these steps using PyTorch's `nn.BatchNorm1d` (https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html) module and validate that the training curves match those from (1.3-1.4) and (2.2)

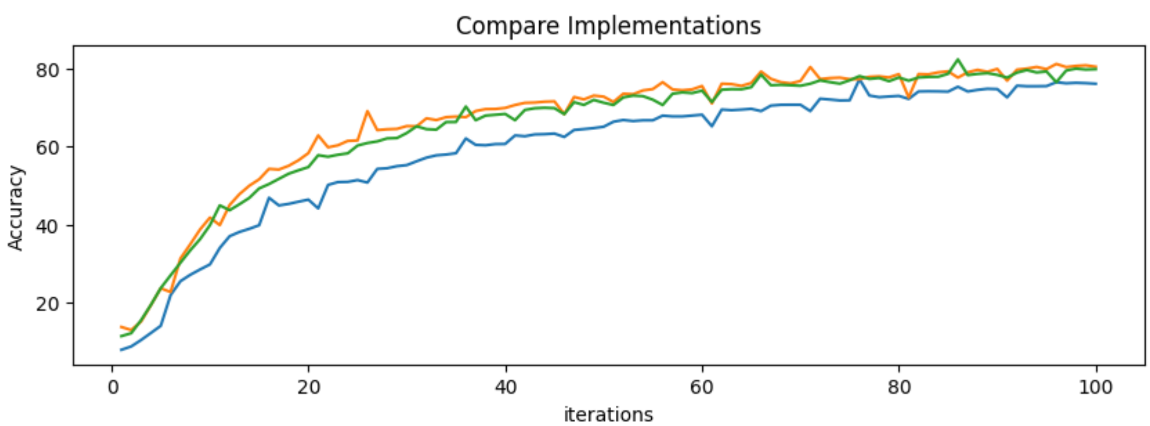
In [ ]:
```
import torch.nn as nn
import torch.nn.functional as F

# 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.bn1 = nn.BatchNorm1d(h1_size)
        self.bn2 = nn.BatchNorm1d(h2_size)
        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.bn1.weight.data.fill_(1)
        self.bn2.weight.data.fill_(1)
        self.bn1.bias.data.zero_()
        self.bn2.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.bn1(x)
        x = torch.relu(x)
        ###
        x = self.linear2(x)
        ### TODO: add batch normalization layer
        x = self.bn2(x)
        ###
        x = F.relu(x)
        return self.linear3(x).view(-1)
```

In [64]:
```
# 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)

train_losses2 = []
train_accuracies2 = []

test_losses = []
test_accuracies = []

for epoch in range(num_epochs):
    train_loss, train_accuracy = train(model, optimizer, train_loader)
    train_losses2.extend(train_loss)
    train_accuracies2.extend(train_accuracy)
    test_loss, test_accuracy = test(model, test_loader)
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)

    # print training progress
    print(
        f'Epoch {epoch+1:02d}: '
        f'train loss {train_loss[-1]:.4f}, train acc {train_accuracy[-
        f'test loss {test_loss:.4f}, test acc {test_accuracy:.2f}%'
        )
```

```
Epoch 01: train loss 2.5951, train acc 23.66%, test loss 2.3597, test
acc 28.28%
Epoch 02: train loss 1.8876, train acc 39.83%, test loss 1.7826, test
acc 42.65%
Epoch 03: train loss 1.5418, train acc 49.27%, test loss 1.4899, test
acc 51.15%
Epoch 04: train loss 1.3612, train acc 54.75%, test loss 1.3070, test
acc 56.75%
Epoch 05: train loss 1.2104, train acc 60.29%, test loss 1.1784, test
acc 61.08%
Epoch 06: train loss 1.1044, train acc 63.51%, test loss 1.0832, test
acc 64.42%
Epoch 07: train loss 1.0293, train acc 66.34%, test loss 1.0059, test
acc 67.08%
Epoch 08: train loss 0.9685, train acc 68.41%, test loss 0.9436, test
acc 69.25%
Epoch 09: train loss 0.9223, train acc 69.89%, test loss 0.8932, test
acc 71.10%
Epoch 10: train loss 0.8794, train acc 71.30%, test loss 0.8517, test
acc 72.55%
Epoch 11: train loss 0.8445, train acc 72.07%, test loss 0.8156, test
acc 73.65%
Epoch 12: train loss 0.7948, train acc 74.38%, test loss 0.7816, test
acc 74.87%
Epoch 13: train loss 0.7714, train acc 75.27%, test loss 0.7542, test
acc 75.70%
Epoch 14: train loss 0.7486, train acc 75.65%, test loss 0.7288, test
acc 76.58%
Epoch 15: train loss 0.7217, train acc 77.05%, test loss 0.7060, test
acc 77.34%
Epoch 16: train loss 0.6946, train acc 77.77%, test loss 0.6841, test
acc 78.21%
Epoch 17: train loss 0.6810, train acc 78.72%, test loss 0.6646, test
acc 78.81%
Epoch 18: train loss 0.6753, train acc 78.45%, test loss 0.6476, test
acc 79.38%
Epoch 19: train loss 0.6467, train acc 79.33%, test loss 0.6326, test
acc 79.87%
Epoch 20: train loss 0.6349, train acc 79.91%, test loss 0.6181, test
acc 80.37%
```

In [84]:
```
### TODO: Visualize training curves
fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for ax, (metric, train_metric, test_metric) in zip(axes, [("loss", tra
    ax.plot(np.linspace(0, len(test_metric), len(train_metric)).astype
    ax.plot(np.arange(len(test_metric)), test_metric, label="test")
    ax.set_xlabel('epoch #')
    ax.legend()
    name = "Loss" if metric == "loss" else "Accuracy"
    ax.set_ylabel(name)
    ax.title.set_text(f'{name} convergence')

# Compare
fig, axes = plt.subplots(1, 1, figsize=(10, 3))

axes.plot(np.arange(len(loss_history))+1, loss_history, label='train_v
axes.plot(np.arange(len(train_accuracies))+1, train_accuracies, label=
axes.plot(np.arange(len(train_accuracies2))+1, train_accuracies2, labe
axes.set_xlabel('iterations')
axes.set_ylabel(name)
axes.title.set_text('Compare Implementations')
plt.show()
```





In [ ]: