dev1num2

February 18, 2019

1 Assigment 1, part2

Link to github : https://github.com/HugoCote/Assignment-1-Part-2

1.0.1 Members of the team :

- Srinivas Venkattaramanujam
- Jean-Philippe Gagnon Fleury
- Ahmadreza Godarzvandchegini
- Hugo Côté

In [1]:
```
# deep learning library
import torch
import torchvision
from   torch.utils.data import DataLoader
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

# to import data
import torchvision.transforms as transforms
import torchvision.datasets.mnist as mnist

# we use torch.cuda.Event(enable_timing=True) to measure time
# if you don't have cuda, you can use instead :
# from timeit import default_timer as timer
# import time

import collections           # for ordered_dictionnary
import torch.nn.init as init # to initialize model

import copy # for copy.deepcopy( ... )

import numpy as np

# to make and display plots
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure

# to format time to strings
import datetime

import math # for ceil ()

import IPython.display # to display .png
```

Some cells could require a long time to evaluate, to warn the user that the evaluation of one such cell is completed, it outputs a sound.

```
In [2]: from IPython.display import Audio
        wave = np.sin(1.5*np.pi*400*np.arange(10000)/10000)
        # the following command line produces sound and is used after cells that
        # require more time to execute :
        Audio(wave, rate=10000, autoplay=True)
```

```
Out[2]: <IPython.lib.display.Audio object>
```

If you did not liked that sound, you should disable it By setting want_lound_warning to false.

```
In [3]: want_lound_warning = False
```

To perfrom the hyper-parameters search, we use the following library : It can be installed with the following command :
- pip install sobol_seq

```
In [4]: # !pip install sobol_seq
        import sobol_seq
```

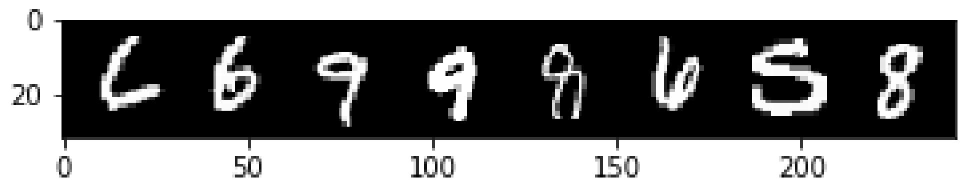
2 Assigment 1, part 2

2.1 Import the data

```
In [5]: transform = transforms.Compose( [transforms.ToTensor()] )
        dl_path  = "./data_mnist"

        train_set = mnist.MNIST( root = dl_path, train=True , download=True, transform =
        transform )
        valid_set = mnist.MNIST( root = dl_path, train=False, download=True, transform =
        transform )
```

Display some samples

```
In [6]: nb_sample = 8

        trainloader = DataLoader(train_set, batch_size = nb_sample, shuffle=True ,
        num_workers=2)
        testloader  = DataLoader(valid_set , batch_size = nb_sample, shuffle=False,
        num_workers=2)

        # functions to show an image

        def imshow(img):
            npimg = img.numpy()
            npimg = (255*npimg).astype(np.uint8) # to be a int in (0,...,255)
            plt.imshow(np.transpose(npimg, (1, 2, 0)))
            plt.show()

        # get some random training images
        dataiter = iter(trainloader)
        images, labels = dataiter.next()

        # show images
        imshow(torchvision.utils.make_grid(images))
        # print labels
        print(' '.join('%5s' % labels[j].item() for j in range(nb_sample)))
```




```
    6      6      9      9      8      6      5      8
```

Print the size of each dataset

```
In [7]: train_dataset_size = train_set.__len__()
        valid_dataset_size  = valid_set.__len__()
        print( "Taining dataset size    : " , train_dataset_size )
        print( "Validation dataset size : " ,valid_dataset_size )
```

```
Taining dataset size    :  60000
Validation dataset size :  10000
```

Set the device to cuda if possible

```
In [8]: device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        print(device)
```

```
cuda:0
```

2.2 Define some modules

2.2.1 the MPL

```
In [9]: class MLP(nn.Module):
            """
            MLP with 2 hidden layer, the parameters h1 and h2 control the size of the model.
            """
            def __init__(self, h1=620,h2=620 ):
                super(MLP, self).__init__()
                self.fc1 = nn.Linear(28 * 28, h1)
                self.fc2 = nn.Linear(h1 , h2)
                self.fc3 = nn.Linear(h2 , 10)

            def forward(self, x):
                x = x.view( x.size()[0] , -1 )
                x = F.relu(self.fc1(x))
                x = F.relu(self.fc2(x))
                x = self.fc3(x)
                return x

            def to_string(self):
                depth_to_string = "The depth of this model is fixed to 3"
                return depth_to_string + self.__doc__
```

2.2.2 The CNN

architecture taken from : https://github.com/MaximumEntropy/welcome_tutorials/tree/pytorch/pytorch

In [52]:
```python
class CNNClassifier(nn.Module):
    """
    CNNClassifier :
    5 Convolutional layers using stride=1, no dilatation and padding to
    assure same convolution, all having :
        - kernel of size 3 doubling the number of feature maps received
          from the previous layer
        - followed by ReLU non-linearity
        - and non-overlapping max pooling with kernel of size 2 (and stride = 2)
        - which means that each layer (made of those 3 steps) :
            - receive as input n  feature maps of size 2m x 2m
            - return  as outpu 2n feature maps of size  m x  m
    With the exeption of :
        - layer 2 has a padding of 2 for the convolution assuring that for the
          rest of the convolutional part the size of the feature maps are
          powers of 2.
        - the last layer does not have a max pooling
    After the convolutional part of the model, the original 1x28x28 input
    picture is now a 256x2x2 vector.
    The 5 conv. layers are followed by two fully connected layer, the first
    having ReLU non-linearity. The parameter size can be chosen to change the
    size of this part of the model.
    For the output of this model to be seen as a probabilie dist., it has to
    be fed to a F.softmax(...,dim=-1)
    """
    def __init__(self, size=472):
        self.layer_size = size
        super(CNNClassifier, self).__init__()
        self.conv = nn.Sequential(
            # Layer 1, input size = 28^2
            nn.Conv2d(in_channels=1, out_channels=16, kernel_size=(3, 3), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),

            # Layer 2, input size = 14^2 -(conv)-> 16^2 -(pool)-> 8^2
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(3, 3), padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),

            # Layer 3, input size = 8^2
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(3, 3), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),

            # Layer 4, input size = 4^2
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(3, 3), padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2),

            # Layer 5, input size = 2^2
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=(3, 3), padding=1),
            nn.ReLU()
        )
        #
        self.fct1 = nn.Linear(4*256, self.layer_size)
        self.fct2 = nn.Linear(self.layer_size,   10 )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size()[0],-1)
        x = F.relu(self.fct1(x))
        x = self.fct2(x)
        return x
```

```
    def to_string(self):
        depth_to_string = "The depth of this model is fixed to 7"
        return depth_to_string + self.__doc__
```

We define a function that computes the number of parameters in a model (and displays its computation)

```
In [53]: def number_of_params( net , display_comp = False ) :
             nb_param  = 0
             depth     = 0 # count the number of different bias
             param_lst = " "
             for i, (key, value) in enumerate( net.state_dict().items() ) :
                 if key.endswith("bias") :
                     depth = depth + 1

                 if i == 0 :
                     param_lst = param_lst + "\n ({:<20}    ".format( key + ")" )
                 else :
                     param_lst = param_lst + "\n ({:<20}  + ".format( key + ")" )


                 nb_param_tmp = 1

                 for j , x in enumerate(value.size()) :
                     if j == 0 :
                         param_lst = param_lst + "{xx}".format( xx = x )
                     else :
                         param_lst = param_lst + "*{xx}".format( xx = x )

                     nb_param_tmp = nb_param_tmp * x

                 nb_param = nb_param + nb_param_tmp

             if display_comp:
                 print( "Number of parameters = " , nb_param , " = ", param_lst  )

             return nb_param, depth
```

Question 1: Come up with a CNN architecture with more or less similar number of parameters as MLP trained in Problem 1 and describe it.

Print the number of parameters in each model with a descrition. We see that the all the model have about the same number of parameters (with a relative difference of 0.33%)

```
In [54]: list_of_models = [ MLP(), CNNClassifier() ]
         nb_of_params = np.zeros(2)
         for i,net in enumerate(list_of_models):
             print( net.__doc__ )
             nb_of_params[i] , _ = number_of_params( net , display_comp = True )

         print("\nRelative difference = " ,
         100*abs(nb_of_params[0]-nb_of_params[1])/nb_of_params[0] , "%")
```

```
    MLP with 2 hidden layer, the parameters h1 and h2 control the size of the model.

Number of parameters =  877930  =
 (fc1.weight)            620*784
 (fc1.bias)            + 620
 (fc2.weight)          + 620*620
 (fc2.bias)            + 620
 (fc3.weight)          + 10*620
```

```
 (fc3.bias)             + 10

    CNNClassifier :
    5 Convolutional layers using stride=1, no dilatation and padding to
    assure same convolution, all having :
        - kernel of size 3 doubling the number of feature maps received
          from the previous layer
        - followed by ReLU non-linearity
        - and non-overlapping max pooling with kernel of size 2 (and stride = 2)
        - which means that each layer (made of those 3 steps) :
            - receive as input n  feature maps of size 2m x 2m
            - return  as outpu 2n feature maps of size  m x  m
    With the exeption of :
        - layer 2 has a padding of 2 for the convolution assuring that for the
          rest of the convolutional part the size of the feature maps are
          powers of 2.
        - the last layer does not have a max pooling
    After the convolutional part of the model, the original 1x28x28 input
    picture is now a 256x2x2 vector.
    The 5 conv. layers are followed by two fully connected layer, the first
    having ReLU non-linearity. The parameter size can be chosen to change the
    size of this part of the model.
    For the output of this model to be seen as a probabilie dist., it has to
    be fed to a F.softmax(...,dim=-1)

Number of parameters =  880850  =
 (conv.0.weight)          16*1*3*3
 (conv.0.bias)          + 16
 (conv.3.weight)        + 32*16*3*3
 (conv.3.bias)          + 32
 (conv.6.weight)        + 64*32*3*3
 (conv.6.bias)          + 64
 (conv.9.weight)        + 128*64*3*3
 (conv.9.bias)          + 128
 (conv.12.weight)       + 256*128*3*3
 (conv.12.bias)         + 256
 (fct1.weight)          + 472*1024
 (fct1.bias)            + 472
 (fct2.weight)          + 10*472
 (fct2.bias)            + 10

Relative difference =  0.3326005490187145 %
```

Test a model To see that there is no bug and that its output has the desired shape

In [13]:
```python
cudanet = CNNClassifier()
_ = cudanet.to(device)
nb_sample = 2
train_loader = DataLoader(train_set, batch_size = nb_sample, shuffle=True ,
num_workers=0)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(cudanet.parameters(), lr=0.00075, momentum=0, weight_decay=0)
want_to_test = False
want_to_test = True
if want_to_test:
    with torch.no_grad() :
        for i, data in enumerate(train_loader, 0):
            # get the inputs
            inputs, labels = data
```

```
            # if using BCE
            # labels = labels.float()
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = cudanet(inputs)
            loss = criterion(outputs, labels)

            print( outputs.size() , labels.size() )
            print( loss )

            break
```

```
torch.Size([2, 10]) torch.Size([2])
tensor(2.3109, device='cuda:0')
```

2.2.3 Training and measuring accuracy algorithms

with some decorators

```
In [14]: # make sound once done, should only be used to wrap a function that returns nothing
         def make_sound(func):
             def wrapper_make_sound(*args, **kwargs):
                 func(*args, **kwargs)
                 wave = np.sin(1.5*np.pi*400*np.arange(10000)/10000)
                 audio = Audio(wave, rate=10000, autoplay=True)
                 if want_lound_warning :
                     return audio
             return wrapper_make_sound

         # measure time with cuda events
         def display_timer(func):
             def wrapper_display_timer(*args, **kwargs):
                 torch.cuda.synchronize()
                 start = torch.cuda.Event(enable_timing=True)
                 end   = torch.cuda.Event(enable_timing=True)
                 start.record()
                 res = func(*args, **kwargs)
                 end.record()
                 torch.cuda.synchronize()
                 print( "Time required = " , start.elapsed_time(end)*0.001 , " s ")
                 return res
             return wrapper_display_timer

         @make_sound
         @display_timer
         def training_phase( net, nb_epoch, criterion, optimizer, avg_loss, accuracy,
         train_loader, state_dict_list ):
             correct      = torch.tensor([0], device = device)
             total        = torch.tensor([0], device = device)
             running_loss = torch.tensor([0.0], dtype=torch.float, device = device)
             for epoch in range( nb_epoch ):  # loop over the dataset multiple times

                 running_loss = 0.0
                 for i, data in enumerate(train_loader, 0):
                     # get the inputs
                     inputs, labels = data
                     inputs, labels = inputs.to(device), labels.to(device)

                     # zero the parameter gradients
                     optimizer.zero_grad()

                     # forward + backward + optimize
                     outputs = net(inputs)
                     loss = criterion(outputs, labels)
                     loss.backward()
```

```
            optimizer.step()

            # compute the correctness of the output labels
            with torch.no_grad() :
                _, predicted = torch.max(outputs.data, 1)
                total   += labels.size(0)
                correct += (predicted == labels).sum()

            # print statistics
            running_loss += loss.item()
        else : # print every epoch
            avg_loss[epoch]    = running_loss / i
            accuracy[epoch,0] = 100 * correct.float() / total.float()
            correct       = torch.tensor([0], device = device)
            total         = torch.tensor([0], device = device)
            running_loss  = torch.tensor([0,0], dtype=torch.float, device = device)
            print( 'epoch = %3d, loss = %.6f , accuracy = %4f'
                          % (epoch + 1, avg_loss[epoch], accuracy[epoch,0] )
                 )
            # save the current model's state_dictionnary
            torch.cuda.synchronize()
            tmp_state_dict = {}
            for k, v in net.state_dict().items():
                tmp_state_dict[k] = v.cpu()
            state_dict_list.append( tmp_state_dict )
            torch.cuda.synchronize()

    else :
        print('Finished Training')

# measure accuracy of a single net, returns the accuracy
def measure_single_accuracy( net, loader ):
    accuracy = 0
    with torch.no_grad():
        correct = torch.tensor([0], device=device)
        total   = torch.tensor([0], device=device)
        for data in loader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total   += labels.size(0)
            correct += (predicted == labels).sum()

        accuracy = 100 * correct.type(torch.FloatTensor) / total.type(torch.FloatTensor)

    return accuracy

# loads state dictionnary from state_dict_list, measure their accuracy, saves the
results in arg accuracy
@display_timer
def measure_accuracy( net, accuracy, train_loader, valid_loader, state_dict_list ,
                        measure_train_accuracy, measure_valid_accuracy ):

    for epoch , tmp_state_dict in enumerate(state_dict_list,0) :

        oldnet = copy.deepcopy( net )
        oldnet.load_state_dict( tmp_state_dict )
        _ = oldnet.to(device)

        # only update desired value
        if measure_train_accuracy:
            accuracy[epoch,0] = measure_single_accuracy( oldnet , train_loader )
        if measure_valid_accuracy:
            accuracy[epoch,1] = measure_single_accuracy( oldnet , valid_loader )

        print('epoch %3d : Accuracy of the network on the validation images: %.2f %% ,
training images %.2f %% '
```

```
                    % ( epoch+1, accuracy[epoch,1] , accuracy[epoch,0] )
                )
```

2.2.4 Plotting functions

In [15]:
```
# plot a two bar charts, accuracy.shape must be n x 2,
# want_log indicates that user wants to save the plot to a file
# filename should not contains the extension of the file
def plot_accuracy_1d( net, lr, bs, accuracy, path_to_save, filename, net_name="",
want_log = False, figsize = (16,10), font_size = 16 ) :
    # data to plot
    n_groups = accuracy.shape[0]
    accuracy_toplot = copy.deepcopy(accuracy).cpu().numpy()

    tests_accuracy = accuracy_toplot[:,1]
    train_accuracy = accuracy_toplot[:,0]

    #
    plt.rcParams.update({'font.size': font_size})
    plt.rcParams["figure.figsize"] = figsize
    # create plot
    fig, ax = plt.subplots()
    index = np.arange(n_groups)
    bar_width = 0.3
    opacity = 0.8

    rects1 = plt.bar(index, tests_accuracy, bar_width,
                     alpha=opacity,
                     color='blue',
                     label='Test set')

    rects2 = plt.bar(index + bar_width, train_accuracy, bar_width,
                     alpha=opacity,
                     color='r',
                     label='Training set')

    eps = 3
    top = min(int( np.ceil(accuracy_toplot.max() + eps)) , 100)
    bot = max(int(np.floor(accuracy_toplot.min() - eps)) , 0  )


    plt.ylim(bot, top)      # set the ylim to bottom, top
    plt.axhline(y=97,color="black")

    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title(net_name + "Comparison between training set and test set accuracy \nduring
the training phase")
    xjump = 1
    plt.xticks( np.arange(0, n_groups+1, step=xjump) , range(0,n_groups+2,xjump) )
    plt.yticks( range(bot,top+1,1) )
    plt.legend()

    if want_log :
        plt.savefig(path_to_save + filename + ".png")
        with open(path_to_save + filename + ".txt",'w+') as f:
            nb_params, depth = number_of_params(net)
            line = "{name} : number of parameters = {n}, depth = {d}. , lr = {lr}, batch
size = {bs}".format(name = net.__doc__, n = nb_params, d=depth, lr=lr, bs = bs)
            f.write(line)
    plt.show()
```

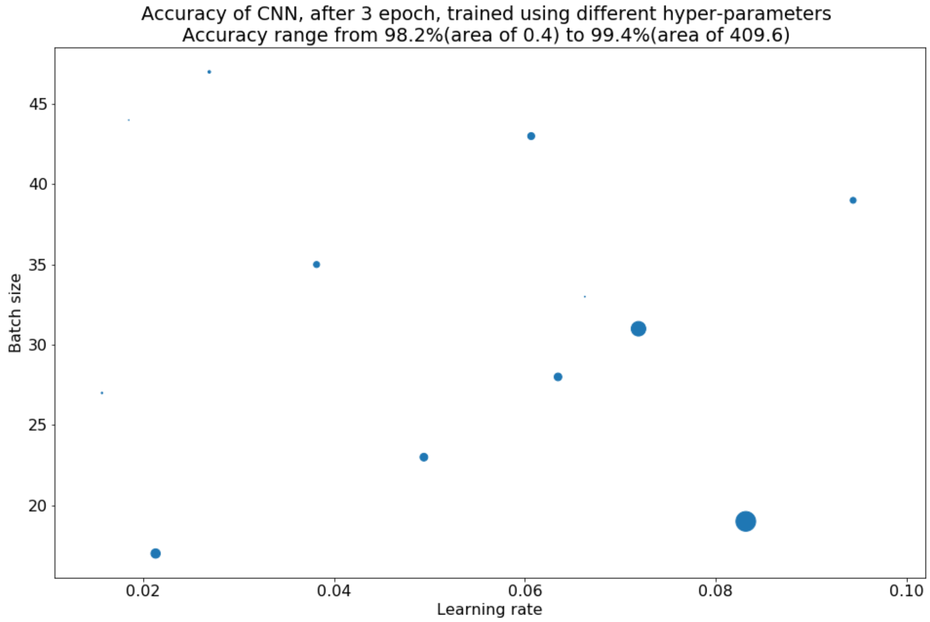
2.2.5 Define some initialization methods

For the purpose of this exercice, we'll only use the glorot initialization if net is an instance of a class inheriting from nn.Module, net.apply(glorot_init) will apply the function glorot_init recursively to itself and all its submodule.

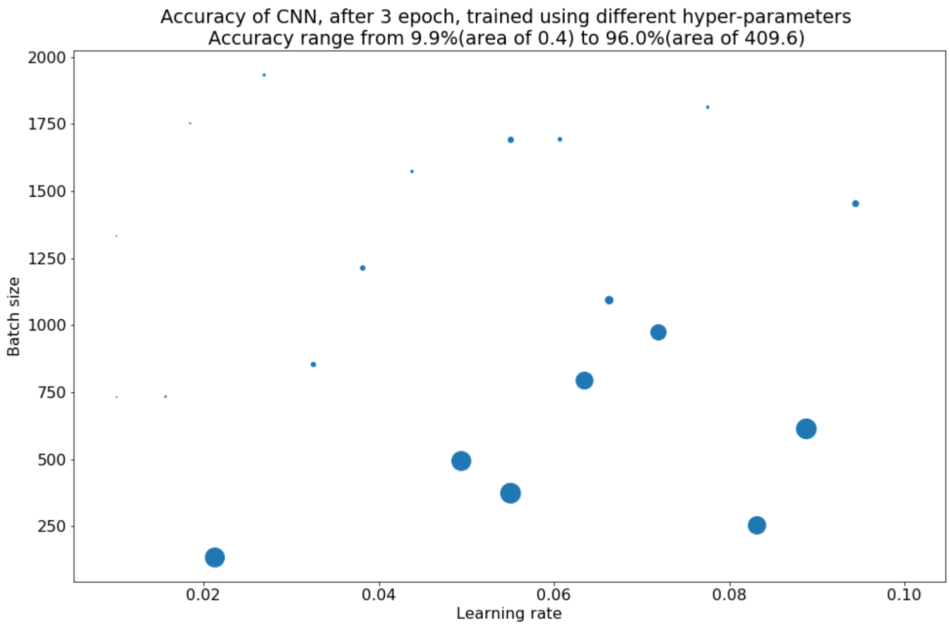
```
In [16]: def glorot_init ( layer ) :
             """
             Weiths are generated from U[-d,d] where d = sqrt(6/(fan_in + fan_out)), biases are
         set to zero
             """
             if type(layer) == nn.Linear or type(layer) == nn.Conv2d :
                 init.xavier_uniform_( layer.weight , gain=1 )
                 layer.bias.data.fill_(0.0)

         def zero_init ( layer ) :
             """Everything is set to zero"""
             if type(layer) == nn.Linear or type(layer) == nn.Conv2d :
                 layer.weight.data.fill_(0.0)
                 layer.bias.data.fill_(0.0)

         def norm_init ( layer ) :
             """Weiths are generated from standard normal, biases are set to zero"""
             if type(layer) == nn.Linear or type(layer) == nn.Conv2d :
                 init.normal_(layer.weight, mean=0, std=1)
                 layer.bias.data.fill_(0.0)
```

Choosing hyper-parameters We previously have made a search to find good hyper-parameters (learning rate and batch size) to train the CNN for 10 epochs. Here is some of the results. For the purpose of saving time, even though the best results have been found with very small batch size (size ~20), we offer a combinason of hyper-parameters with large batch size.

```
In [17]: # Here's what we add when we runned the notebook :
         # Searching hyper-parameters space using a sobol sequence using :
         # start      = 2030      (the starting point in the sequence)
         # nb_points = 20         (number of points to evaluate)
         # lr_min = 0.01          (lower bound for learning rate)
         # lr_max = 0.1           (upper bound for learning rate)
         # batch_size_min =  2*8 (lower bound for batch size)
         # batch_size_max =  6*8 (upper bound for batch size)
         loading_path = "./output/2019February15_AM11H37M.png"
         IPython.display.display(IPython.display.Image(filename=loading_path))
```




```
In [18]: # Searching hyper-parameters space using a sobol sequence (see later) using :
         # start      = 2030      (the starting point in the sequence)
         # nb_points = 20         (number of points to evaluate)
         # lr_min = 0.01          (lower bound for learning rate)
         # lr_max = 0.1           (upper bound for learning rate)
         # batch_size_min = 2*64 (lower bound for batch size)
         # batch_size_max = 32*64(upper bound for batch size)
         loading_path = "./output/2019February15_PM12H19M.png"
         IPython.display.display(IPython.display.Image(filename=loading_path))
```




```
In [19]: # instantiate CNNClassifier and load it to the gpu if possible
         net_cnn = CNNClassifier()
         net_cnn.apply( glorot_init )
         _ = net_cnn.to(device)
```

```
In [20]: # define the loss function as the cross entropy and choose a learning rate that works
         well
         # Restraining ourselves to small batch size, the search found those hyper-parameters :
         # lr = 0.0831689
         # training_batch_size = 19
         #                           with large batch size, the search found those hyper-
         parameters :
         # lr = 0.0505483
         # training_batch_size = 373

         criterion = nn.CrossEntropyLoss()
         lr = 0.0831689
         optimizer = optim.SGD(net_cnn.parameters(), lr=lr, momentum=0.0, weight_decay=0)
         nb_epoch  = 10

         training_batch_size = 19
         train_loader = DataLoader(train_set, batch_size=training_batch_size,shuffle=True,
         num_workers=2)
         cnn_state_dict_list = list()        # we save (all) the intermediate state of the model
         during the learning phase

         # average loss across epoch
         cnn_avg_loss     = torch.empty(nb_epoch  , dtype=torch.float, device = device)
         # accuracy[i, 0 (resp. 1)] is the training (reps. validation) accuracy of the net at
         epoch i
         cnn_accuracy     = torch.empty(nb_epoch,2, dtype=torch.float, device = device)
```

Just to see that all is fine : we make sure that in its current state the net is randomly guessing and has around 10% accuracy.

```
In [21]: valid_batch_size = 32*64
         valid_loader   = DataLoader(valid_set, batch_size=valid_batch_size,shuffle=True,
         num_workers=2)
         start_accuracy = measure_single_accuracy( net_cnn, valid_loader )
         print("Accuracy at initialization : " , start_accuracy.item() , "%" )
```

```
Accuracy at initialization :  11.369999885559082 %
```

```
In [22]: training_phase( net_cnn, nb_epoch, criterion, optimizer, cnn_avg_loss, cnn_accuracy,
         train_loader, cnn_state_dict_list )
```

```
epoch =   1, loss = 0.229995 , accuracy = 92.431664
epoch =   2, loss = 0.051981 , accuracy = 98.379997
epoch =   3, loss = 0.034742 , accuracy = 98.904999
epoch =   4, loss = 0.026171 , accuracy = 99.201668
epoch =   5, loss = 0.021411 , accuracy = 99.364998
epoch =   6, loss = 0.015973 , accuracy = 99.489998
epoch =   7, loss = 0.015118 , accuracy = 99.528336
epoch =   8, loss = 0.012356 , accuracy = 99.621666
epoch =   9, loss = 0.010727 , accuracy = 99.653336
epoch =  10, loss = 0.007961 , accuracy = 99.758331
Finished Training
Time required =  583.0525  s
```

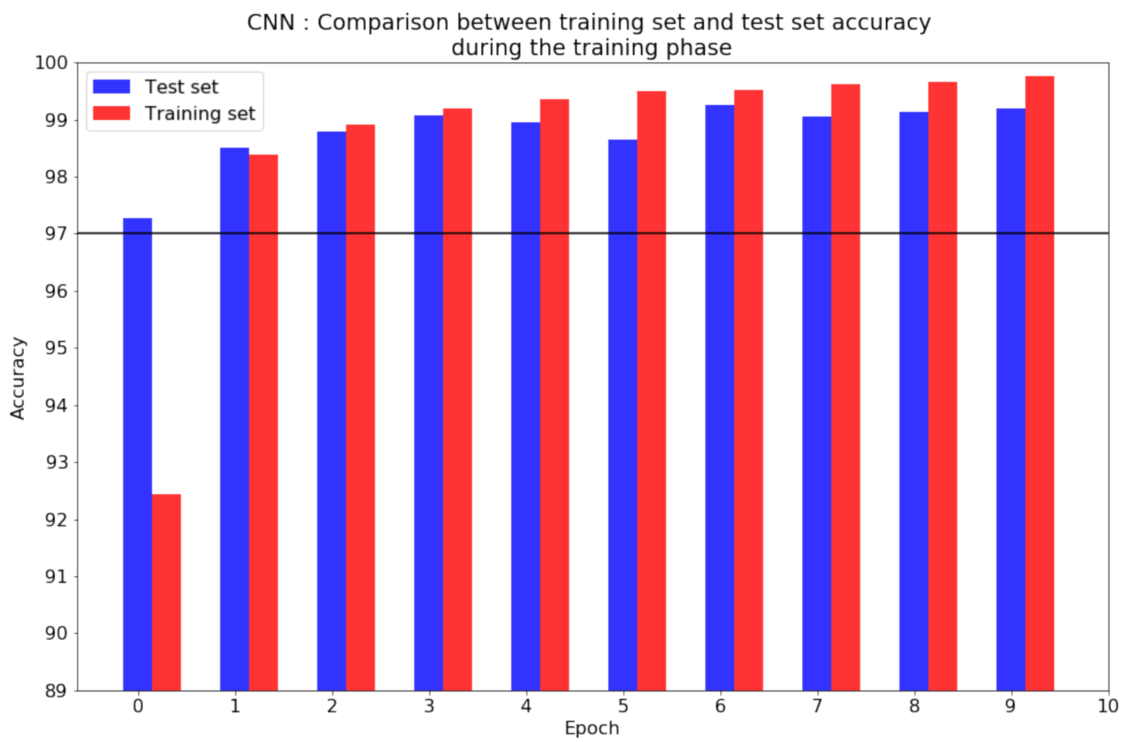
Measure the accuracy of the net on the dataset(s) across epoch We retrieve every state dictionary on the list, load it to the net
and compute the accuracy on the chosen dataset(s).

```
In [23]: batch_size = 32*64
         train_loader = DataLoader(train_set, batch_size = batch_size, shuffle=False ,
         num_workers=2)
         valid_loader = DataLoader(valid_set , batch_size = batch_size, shuffle=False ,
         num_workers=2)

         measure_train_accuracy = False # this is already computed
         measure_valid_accuracy = True  #

         measure_accuracy( net_cnn , cnn_accuracy, train_loader, valid_loader,
         cnn_state_dict_list ,
                                   measure_train_accuracy, measure_valid_accuracy )
```

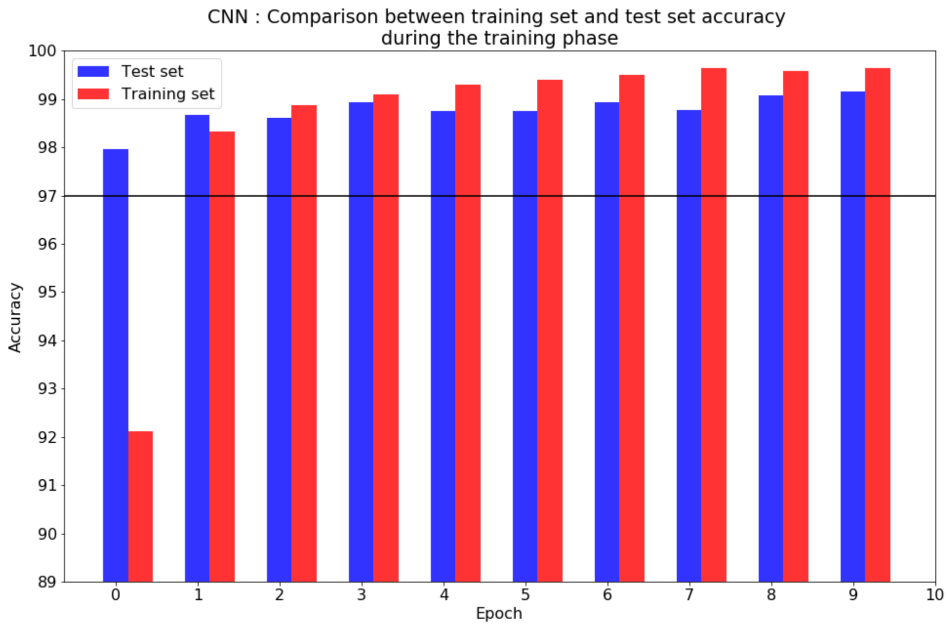
```
epoch   1 : Accuracy of the network on the validation images: 97.28 % , training
images 92.43 %
epoch   2 : Accuracy of the network on the validation images: 98.51 % , training
images 98.38 %
epoch   3 : Accuracy of the network on the validation images: 98.80 % , training
images 98.90 %
epoch   4 : Accuracy of the network on the validation images: 99.07 % , training
images 99.20 %
epoch   5 : Accuracy of the network on the validation images: 98.95 % , training
images 99.36 %
epoch   6 : Accuracy of the network on the validation images: 98.64 % , training
images 99.49 %
epoch   7 : Accuracy of the network on the validation images: 99.25 % , training
images 99.53 %
epoch   8 : Accuracy of the network on the validation images: 99.06 % , training
images 99.62 %
epoch   9 : Accuracy of the network on the validation images: 99.14 % , training
```

```
images 99.65 %
epoch  10 : Accuracy of the network on the validation images: 99.20 % , training
images 99.76 %
Time required =  13.728021484375  s
```

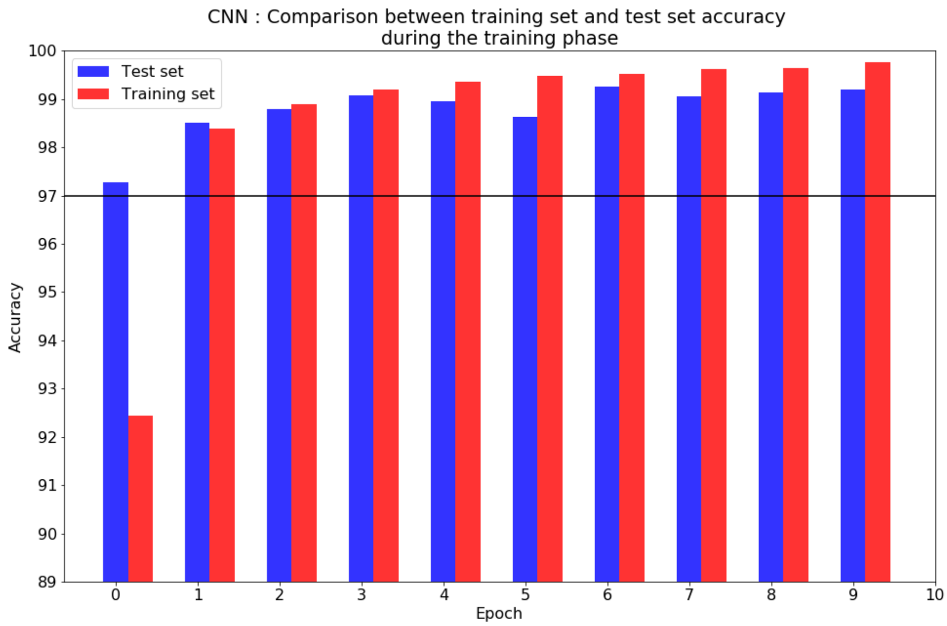
In [24]:
```
# Plot the accuracy
want_log     = True
path_to_save = "./output/"
filename     = datetime.datetime.now().strftime("%Y%B%d_%p%IH%MM")
plot_accuracy_1d( net_cnn, lr, training_batch_size, cnn_accuracy, path_to_save,
filename, "CNN : ", want_log )
```



In [25]:
```
# Here's what we add when we runned the notebook using large batch size:
loading_path = "./output/2019February15_PM12H02M.png"
IPython.display.display(IPython.display.Image(filename=loading_path))
```




```
In [27]: # Here's what we add when we runned the notebook using small batch size:
         loading_path = "./output/2019February16_PM08H46M.png"
         IPython.display.display(IPython.display.Image(filename=loading_path))
```



3 Train some multi-layer perceptron's (MLP's)

What we plan to do :

a) We generate a pair (learning rate, batch size)

b) We train an mlp for 3 epoch using these hyper-parameters

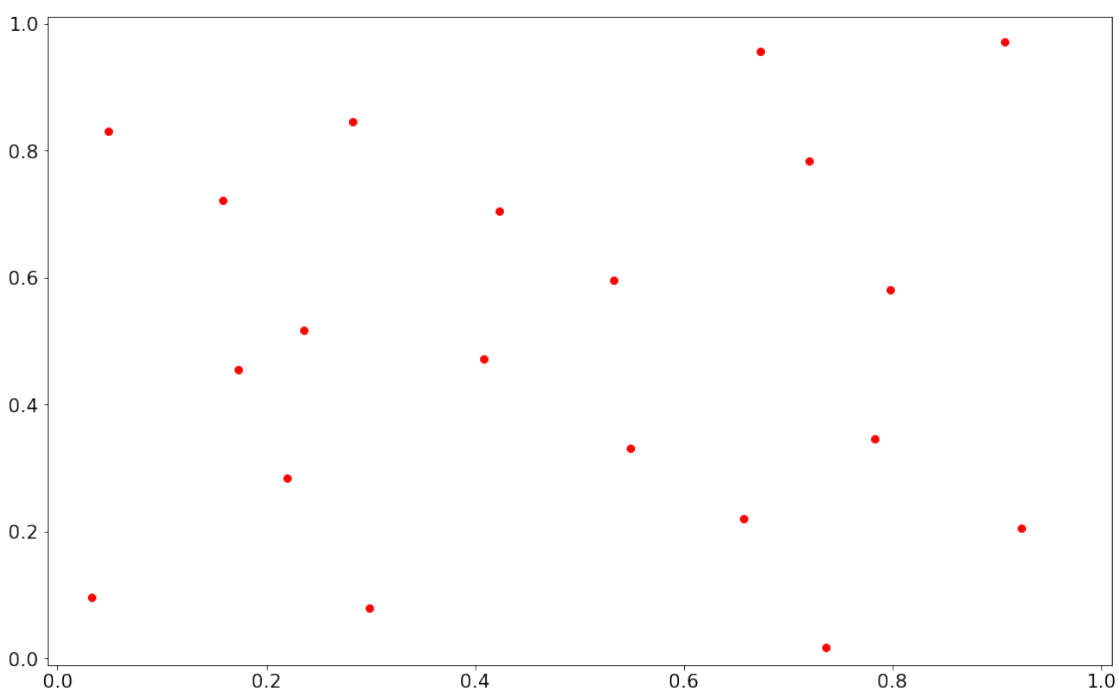
c) We pick the mlp that has the highest accuracy on the validation dataset

d) We train another mlp with the same hyper-parameters for 10 epoch

e) We compare the performance of this mlp against the cnn

Here's vizualisation of successive points from a 2d sobol sequence

In [28]:
```
# Vizualize a sobol sequence
#seq = sobol_seq.i4_sobol_generate(2,32)
start     = 3030
nb_points = 20

seq = np.empty((nb_points,2))
end       = start + nb_points
for i,j in enumerate(range(start,end,1)) :
    seq[i,:] ,_ = sobol_seq.i4_sobol(2,j)

plt.plot(seq[:,0],seq[:,1], 'ro')
eps = 0.01
plt.axis([0 - eps , 1 + eps , 0 - eps , 1 + eps ])
plt.show()
```



```
In [29]: nb_epoch  = 3

         # current mlp with the best performance on the validation set, on its last epoch
         acc_max = 0
         idx_max = 0

         # we save (all) the intermediate state of the model during the learning phase, for each
         mlp
         state_dict_dict = dict()

         # search around a good learning rate : 0.055
         # lr_min = 0.02
         # lr_max = 0.10
         lr_min = 0.001
         lr_max = 0.1

         # search around a good batch size : 2*8
         batch_size_min =  2*8
         batch_size_max =  6*8
         # batch_size_min =  2 *64
         # batch_size_max =  32*64

         #
         valid_batch_size = 32*64
         valid_loader = DataLoader(train_set, batch_size=valid_batch_size,shuffle=False,
         num_workers=2)

         # we started from 1030 for mlp and from 2030 for cnn
         start     = 1030
         nb_points = 20
         valid_acc = torch.empty(nb_points)
         seq = np.empty((nb_points,2))
         end       = start + nb_points
         for i,j in enumerate(range(start,end,1)) :
             hyperparam_point ,_ = sobol_seq.i4_sobol(2,j)
             lr, batch_size      = hyperparam_point
             # take the point in the unitary cube and map it to the desired box
             lr         = lr*(lr_max-lr_min) +lr_min
             batch_size = math.ceil(batch_size*(batch_size_max-batch_size_min)+batch_size_min)

             if i != 0 :
                 del net_mlp

             net_mlp = MLP()
             # net_mlp = CNNClassifier() # in order to search for good CNN parameters
             net_mlp.apply( glorot_init )
             _ = net_mlp.to(device)

             criterion = nn.CrossEntropyLoss()
             optimizer = optim.SGD(net_mlp.parameters(), lr=lr, momentum=0.0, weight_decay=0)

             train_batch_size = batch_size
             train_loader = DataLoader(train_set, batch_size=train_batch_size,shuffle=True ,
         num_workers=2)
             state_dict_list = list() # we save (all) the intermediate state of the model during
         the learning phase, for one  mlp

             # average loss across epoch
             avg_loss     = torch.empty(nb_epoch   , dtype=torch.float, device = device)
             # accuracy[i, 0 (resp. 1)] is the training (reps. validation) accuracy of the net at
         epoch i
             accuracy     = torch.empty(nb_epoch,2, dtype=torch.float, device = device)

             # print hyper-parameters
             print("point no. {i}, lr = {lr}, batch size = {batch_size}".format(i=i,
         lr=lr,batch_size=batch_size))

             # we dump output to disable sound
```

```
            torch.cuda.synchronize()
            _ = training_phase( net_mlp, nb_epoch, criterion, optimizer, avg_loss, accuracy,
        train_loader, state_dict_list )
            state_dict_dict[i] = [[lr,batch_size],state_dict_list,avg_loss,accuracy]


            valid_accuracy = measure_single_accuracy(net_mlp,valid_loader)
            valid_acc[i] = valid_accuracy
            if valid_accuracy > acc_max :
                acc_max = valid_accuracy
                idx_max = i
            torch.cuda.synchronize()

        print("####################")
        print("best net found : {i} , with validation accuracy =
        {va}".format(i=idx_max,va=acc_max.item()))
        Audio(wave, rate=10000, autoplay=True)
```

```
point no. 0, lr = 0.06302001953125, batch size = 25
epoch =   1, loss = 0.260214 , accuracy = 92.398331
epoch =   2, loss = 0.106714 , accuracy = 96.803337
epoch =   3, loss = 0.071194 , accuracy = 97.860001
Finished Training
Time required =  58.16783203125  s
point no. 1, lr = 0.01352001953125, batch size = 41
epoch =   1, loss = 0.545274 , accuracy = 86.381668
epoch =   2, loss = 0.266535 , accuracy = 92.296669
epoch =   3, loss = 0.215279 , accuracy = 93.930000
Finished Training
Time required =  38.16459765625  s
point no. 2, lr = 0.019707519531250002, batch size = 23
epoch =   1, loss = 0.379862 , accuracy = 89.691666
epoch =   2, loss = 0.181625 , accuracy = 94.846664
epoch =   3, loss = 0.130913 , accuracy = 96.246666
Finished Training
Time required =  63.173500000000004  s
point no. 3, lr = 0.06920751953125, batch size = 39
epoch =   1, loss = 0.298062 , accuracy = 91.410004
epoch =   2, loss = 0.125626 , accuracy = 96.276665
epoch =   3, loss = 0.084634 , accuracy = 97.528336
Finished Training
Time required =  39.3976328125  s
point no. 4, lr = 0.09395751953125, batch size = 31
epoch =   1, loss = 0.249312 , accuracy = 92.574997
epoch =   2, loss = 0.098090 , accuracy = 97.028336
epoch =   3, loss = 0.064959 , accuracy = 98.058334
Finished Training
Time required =  51.217132812500004  s
point no. 5, lr = 0.04445751953125, batch size = 47
epoch =   1, loss = 0.362966 , accuracy = 90.129997
epoch =   2, loss = 0.171145 , accuracy = 95.059998
epoch =   3, loss = 0.122349 , accuracy = 96.481667
Finished Training
Time required =  32.7171796875  s
point no. 6, lr = 0.03208251953125, batch size = 27
epoch =   1, loss = 0.338921 , accuracy = 90.501663
epoch =   2, loss = 0.151898 , accuracy = 95.570000
epoch =   3, loss = 0.104915 , accuracy = 96.898331
Finished Training
Time required =  58.03594921875  s
point no. 7, lr = 0.08158251953125001, batch size = 43
```

```
epoch =   1, loss = 0.288099 , accuracy = 91.620003
epoch =   2, loss = 0.118830 , accuracy = 96.503334
epoch =   3, loss = 0.080092 , accuracy = 97.643333
Finished Training
Time required =  37.3554140625  s
point no. 8, lr = 0.05683251953125, batch size = 19
epoch =   1, loss = 0.245502 , accuracy = 92.864998
epoch =   2, loss = 0.098671 , accuracy = 97.008331
epoch =   3, loss = 0.064808 , accuracy = 98.041664
Finished Training
Time required =  78.2053203125  s
point no. 9, lr = 0.007332519531250001, batch size = 35
epoch =   1, loss = 0.671407 , accuracy = 83.989998
epoch =   2, loss = 0.307693 , accuracy = 91.341667
epoch =   3, loss = 0.257016 , accuracy = 92.686668
Finished Training
Time required =  42.7760546875  s
point no. 10, lr = 0.01042626953125, batch size = 20
epoch =   1, loss = 0.452525 , accuracy = 88.228333
epoch =   2, loss = 0.222902 , accuracy = 93.723335
epoch =   3, loss = 0.170617 , accuracy = 95.106667
Finished Training
Time required =  75.753125  s
point no. 11, lr = 0.05992626953125001, batch size = 36
epoch =   1, loss = 0.297520 , accuracy = 91.426666
epoch =   2, loss = 0.127538 , accuracy = 96.258331
epoch =   3, loss = 0.086453 , accuracy = 97.496666
Finished Training
Time required =  41.14410546875  s
point no. 12, lr = 0.08467626953125, batch size = 28
epoch =   1, loss = 0.244769 , accuracy = 92.785004
epoch =   2, loss = 0.098322 , accuracy = 97.028336
epoch =   3, loss = 0.064640 , accuracy = 98.044998
Finished Training
Time required =  56.77841796875  s
point no. 13, lr = 0.035176269531250005, batch size = 44
epoch =   1, loss = 0.390159 , accuracy = 89.345001
epoch =   2, loss = 0.189044 , accuracy = 94.639999
epoch =   3, loss = 0.136736 , accuracy = 96.086670
Finished Training
Time required =  35.784953125  s
point no. 14, lr = 0.04755126953125, batch size = 32
epoch =   1, loss = 0.319307 , accuracy = 90.919998
epoch =   2, loss = 0.137379 , accuracy = 95.981667
epoch =   3, loss = 0.092343 , accuracy = 97.398331
Finished Training
Time required =  48.90377734375  s
point no. 15, lr = 0.09705126953125001, batch size = 48
epoch =   1, loss = 0.280003 , accuracy = 91.858330
epoch =   2, loss = 0.114078 , accuracy = 96.606667
epoch =   3, loss = 0.076758 , accuracy = 97.730003
Finished Training
Time required =  33.7703359375  s
point no. 16, lr = 0.07230126953125, batch size = 24
epoch =   1, loss = 0.251128 , accuracy = 92.446663
epoch =   2, loss = 0.100846 , accuracy = 97.003334
epoch =   3, loss = 0.065025 , accuracy = 98.021667
Finished Training
Time required =  65.03314453125  s
```

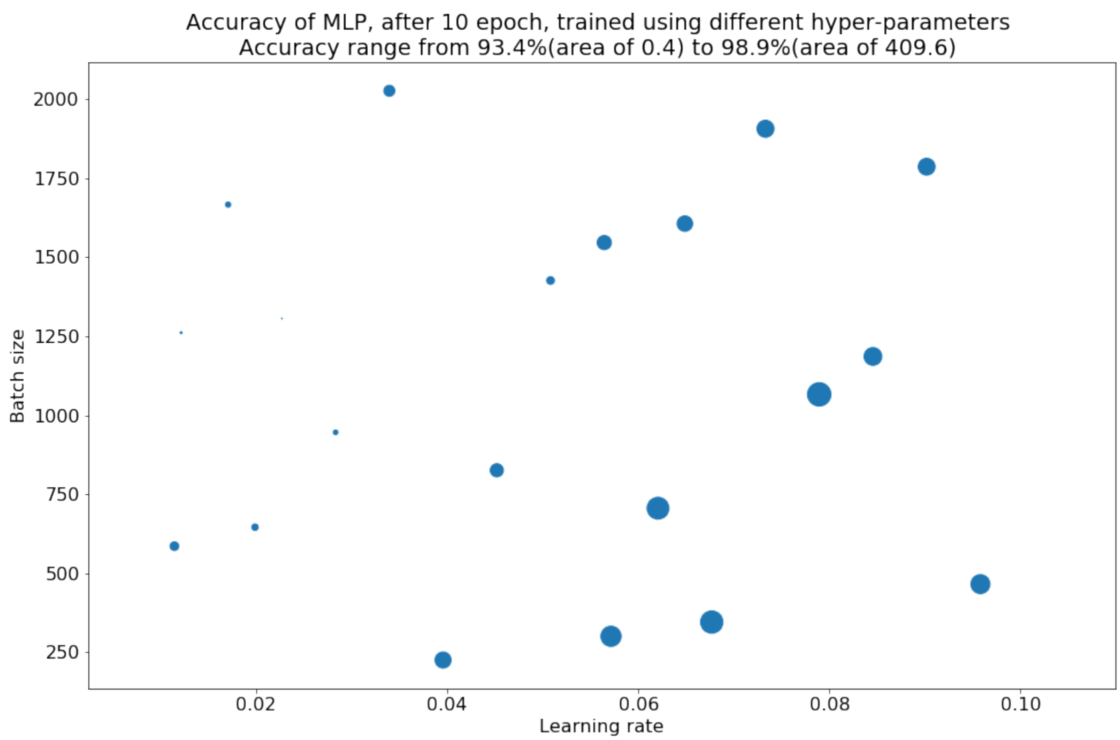
```
point no. 17, lr = 0.02280126953125, batch size = 40
epoch =   1, loss = 0.443788 , accuracy = 88.321663
epoch =   2, loss = 0.218065 , accuracy = 93.841667
epoch =   3, loss = 0.165185 , accuracy = 95.346664
Finished Training
Time required =  39.81326171875  s
point no. 18, lr = 0.01661376953125, batch size = 26
epoch =   1, loss = 0.420921 , accuracy = 88.821663
epoch =   2, loss = 0.203506 , accuracy = 94.161667
epoch =   3, loss = 0.151548 , accuracy = 95.696663
Finished Training
Time required =  57.12751953125  s
point no. 19, lr = 0.06611376953125, batch size = 42
epoch =   1, loss = 0.307322 , accuracy = 91.239998
epoch =   2, loss = 0.132696 , accuracy = 96.053337
epoch =   3, loss = 0.089816 , accuracy = 97.394997
Finished Training
Time required =  37.3833671875  s
#####################
best net found : 4 , with validation accuracy = 98.9183349609375
```

Out[29]: <IPython.lib.display.Audio object>

Display the results from the search

In [31]:
```
# taken from
# https://matplotlib.org/gallery/shapes_and_collections/scatter.html#sphx-glr-gallery-
shapes-and-collections-scatter-py

# search around a good learning rate : 0.055
lr_min = 0.01
lr_max = 0.1

# search around a good batch size : 2*8
# batch_size_min =  2*8
# batch_size_max =  6*8
batch_size_min =  2 *64
batch_size_max =  32*64

# Fixing start state for reproducibility
start     = 4030
nb_points = 20
end       = start + nb_points
_x = torch.empty(nb_points)
_y = torch.empty(nb_points)
for i,j in enumerate(range(start,end,1)) :
    hyperparam_point ,_ = sobol_seq.i4_sobol(2,j)
    lr, batch_size      = hyperparam_point
    # take the point in the unitary cube and map it to the desired box
    lr         = lr*(lr_max-lr_min) +lr_min
    batch_size = math.ceil(batch_size*(batch_size_max-batch_size_min)+batch_size_min)
    _x[i] = lr
    _y[i] = batch_size

N = nb_points
_val = valid_acc
val = _val.numpy()
val = val - val.min()
val = val / val.max()
x = _x.numpy()
y = _y.numpy()
# colors = np.ones(N)*(0.2)
```

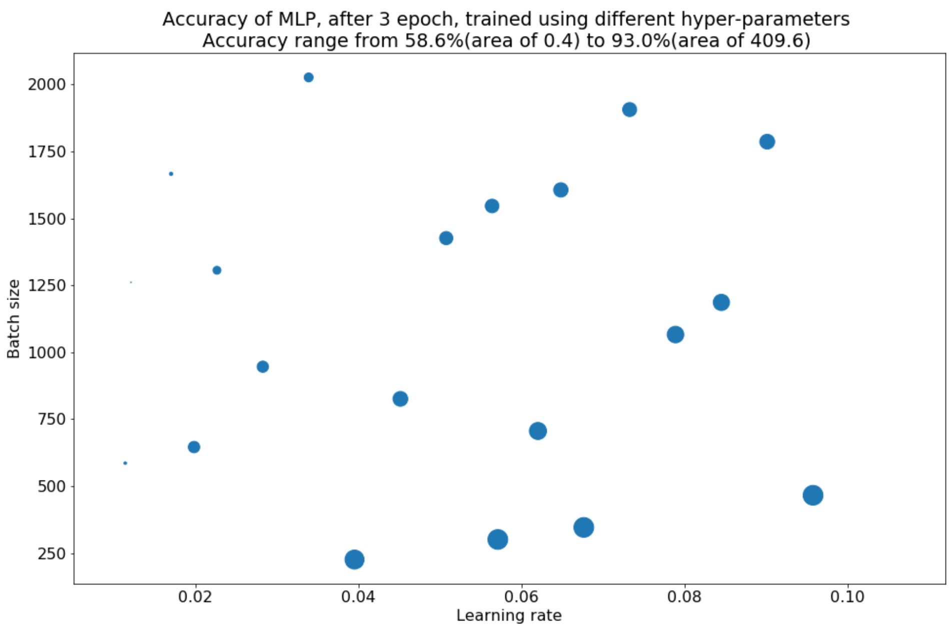
```
area = 0.4*(1+val)**10

str_title1 = "Accuracy of MLP, after {epoch} epoch, trained using different hyper-
parameters \n".format(epoch=nb_epoch)
str_title2 = "Accuracy range from {min:.{prec}f}%(area of {rmin:.{prec}f}) to
{max:.{prec}f}%(area of {rmax:.{prec}f}) ".format(
      min  = torch.min(valid_acc).item(),
      rmin = area.min(),
      max  = torch.max(valid_acc).item(),
      rmax = area.max(),
      prec = 1
    )
plt.rcParams.update({'font.size': 16})
plt.rcParams["figure.figsize"] = (16,10)

plt.title(str_title1+str_title2)
plt.xlabel('Learning rate')
plt.ylabel('Batch size')

plt.scatter(x, y, s=area)
want_log     = True
path_to_save = "./output/"
filename     = datetime.datetime.now().strftime("%Y%B%d_%p%IH%MM")
if want_log :
        plt.savefig(path_to_save + filename + ".png")
plt.show()
```



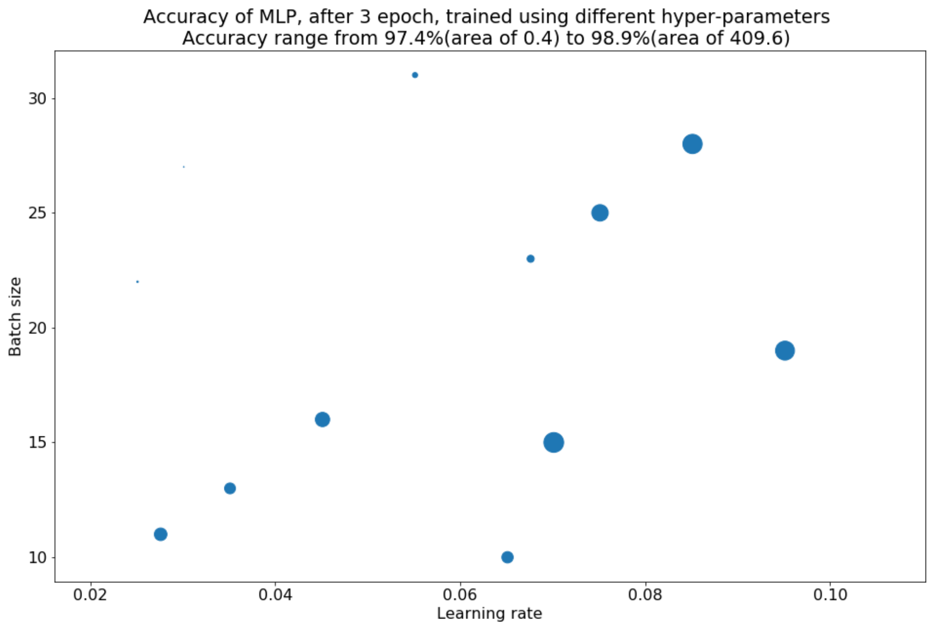
What we had when we runned the notebook

```
In [30]: # Here's what we add when we runned the notebook :
         # Searching hyper-parameters space using a sobol sequence using :
         # start     = 4030      (the starting point in the sequence)
```

```
# nb_points = 20         (number of points to evaluate)
# lr_min = 0.01          (lower bound for learning rate)
# lr_max = 0.1           (upper bound for learning rate)
# batch_size_min =   2*64(lower bound for batch size)
# batch_size_max =  32*64(upper bound for batch size)
loading_path = "./output/2019February15_PM12H36M.png"
IPython.display.display(IPython.display.Image(filename=loading_path))
```




```
In [31]: # Here's what we add when we runned the notebook :
         # Searching hyper-parameters space using a sobol sequence using :
         # start      = 2030       (the starting point in the sequence)
         # nb_points = 12         (number of points to evaluate)
         # lr_min = 0.01          (lower bound for learning rate)
         # lr_max = 0.1           (upper bound for learning rate)
         # batch_size_min =  2*8 (lower bound for batch size)
         # batch_size_max =  6*8 (upper bound for batch size)
         loading_path = "./output/2019February12_PM05H03M.png"
         IPython.display.display(IPython.display.Image(filename=loading_path))
```



3.0.1 Take what seems to be the best choice of hyper-parameters and train an mlp

Training

In [30]:
```
# retrieve the best performing mlp
# In our run, we got (with small batch size):
#                       - lr_mlp = 0.070 (learning rate)
#                       - bs_mlp = 15     (batch size)
#                     (with large batch size):
#                       - lr_mlp = 0.0953835 (learning rate)
#                       - bs_mlp = 466       (batch size)

(lr_mlp,bs_mlp),mlp_dict,mlp_avg_loss,mlp_accuracy = state_dict_dict[idx_max]
net_best_mlp = MLP()
_ = net_best_mlp.to(device)

#
nb_epoch  = 10
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net_best_mlp.parameters(), lr=lr_mlp, momentum=0.0,
weight_decay=0)
train_loader = DataLoader(train_set, batch_size=bs_mlp,shuffle=True, num_workers=2)
mlp_dict = list()
mlp_avg_loss     = torch.empty(nb_epoch  , dtype=torch.float, device = device)
mlp_accuracy     = torch.empty(nb_epoch,2, dtype=torch.float, device = device)

training_phase( net_best_mlp, nb_epoch, criterion, optimizer, mlp_avg_loss,
mlp_accuracy, train_loader, mlp_dict )
```

```
epoch =   1, loss = 0.339883 , accuracy = 90.160004
epoch =   2, loss = 0.116079 , accuracy = 96.491669
epoch =   3, loss = 0.076300 , accuracy = 97.629997
epoch =   4, loss = 0.053927 , accuracy = 98.316666
```

```
epoch =   5, loss = 0.039736 , accuracy = 98.785004
epoch =   6, loss = 0.030245 , accuracy = 99.089996
epoch =   7, loss = 0.022095 , accuracy = 99.364998
epoch =   8, loss = 0.016349 , accuracy = 99.526665
epoch =   9, loss = 0.011686 , accuracy = 99.688332
epoch =  10, loss = 0.008098 , accuracy = 99.820000
Finished Training
Time required =  160.07575  s
```

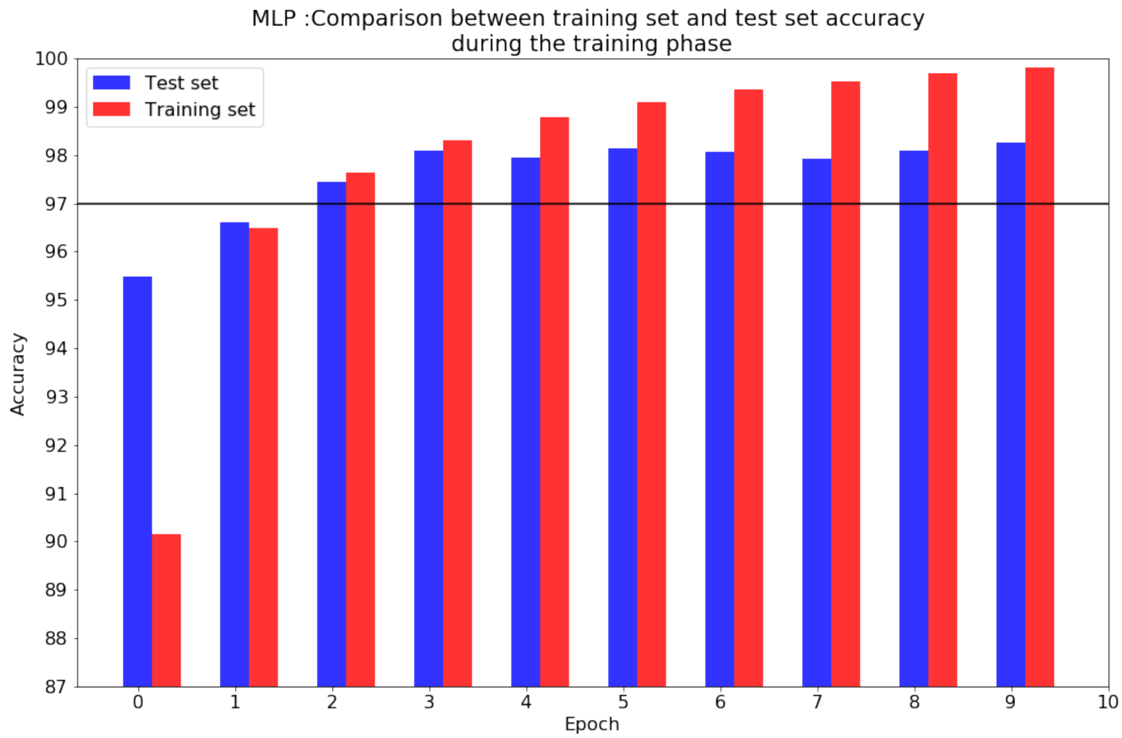
Measure its accuracy

In [32]:
```
measure_train_accuracy = False # this is already computed
measure_valid_accuracy = True  #

batch_size = 16*64
valid_loader = DataLoader(valid_set , batch_size = batch_size, shuffle=False ,
num_workers=2)

measure_accuracy( net_best_mlp , mlp_accuracy, train_loader, valid_loader, mlp_dict ,
                     measure_train_accuracy, measure_valid_accuracy )
```

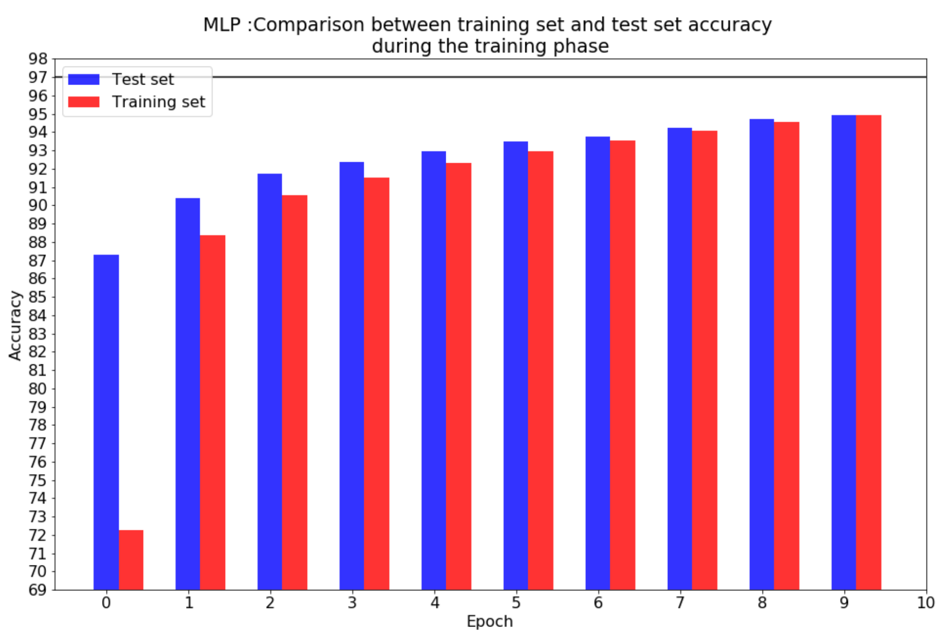
```
epoch   1 : Accuracy of the network on the validation images: 95.49 % , training
images 90.16 %
epoch   2 : Accuracy of the network on the validation images: 96.62 % , training
images 96.49 %
epoch   3 : Accuracy of the network on the validation images: 97.44 % , training
images 97.63 %
epoch   4 : Accuracy of the network on the validation images: 98.08 % , training
images 98.32 %
epoch   5 : Accuracy of the network on the validation images: 97.95 % , training
images 98.79 %
epoch   6 : Accuracy of the network on the validation images: 98.13 % , training
images 99.09 %
epoch   7 : Accuracy of the network on the validation images: 98.07 % , training
images 99.36 %
epoch   8 : Accuracy of the network on the validation images: 97.93 % , training
images 99.53 %
epoch   9 : Accuracy of the network on the validation images: 98.08 % , training
images 99.69 %
epoch  10 : Accuracy of the network on the validation images: 98.26 % , training
images 99.82 %
Time required =  12.8307958984375  s
```

Plot the accuracy of mlp across epoch

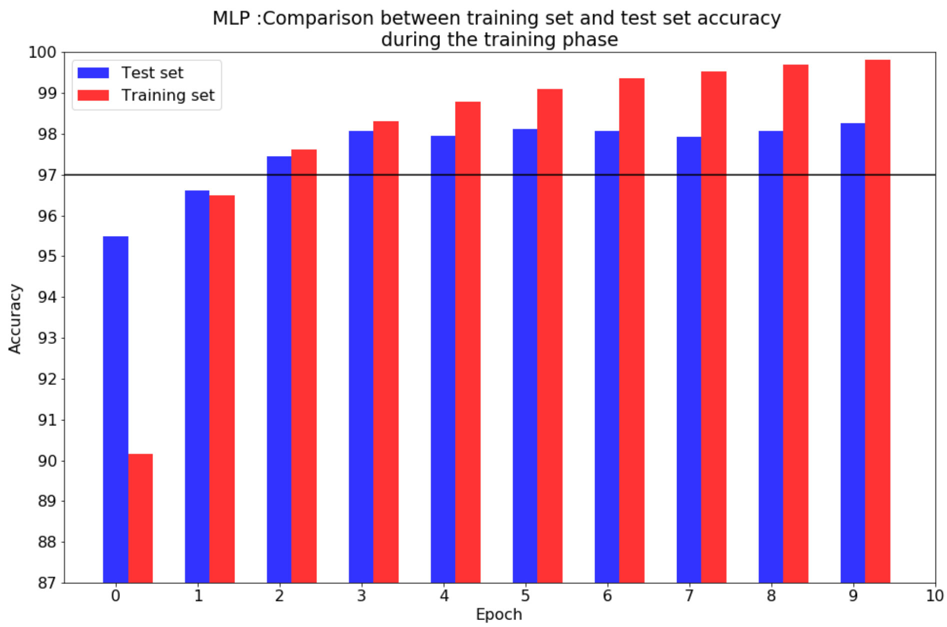
In [34]:
```
# Plot the accuracy
want_log     = False
path_to_save = "./output/"
filename     = datetime.datetime.now().strftime("%Y%B%d_%p%IH%MM")
# plot_accuracy( mlp_accuracy, want_log , path_to_save, filename )
plot_accuracy_1d( net_best_mlp, lr_mlp, bs_mlp, mlp_accuracy, path_to_save, filename ,
"MLP :", want_log)
```




```
In [35]: # Here's what we add when we runned the notebook :
         loading_path = "./output/2019February15_PM12H39M.png"
         IPython.display.display(IPython.display.Image(filename=loading_path))
```



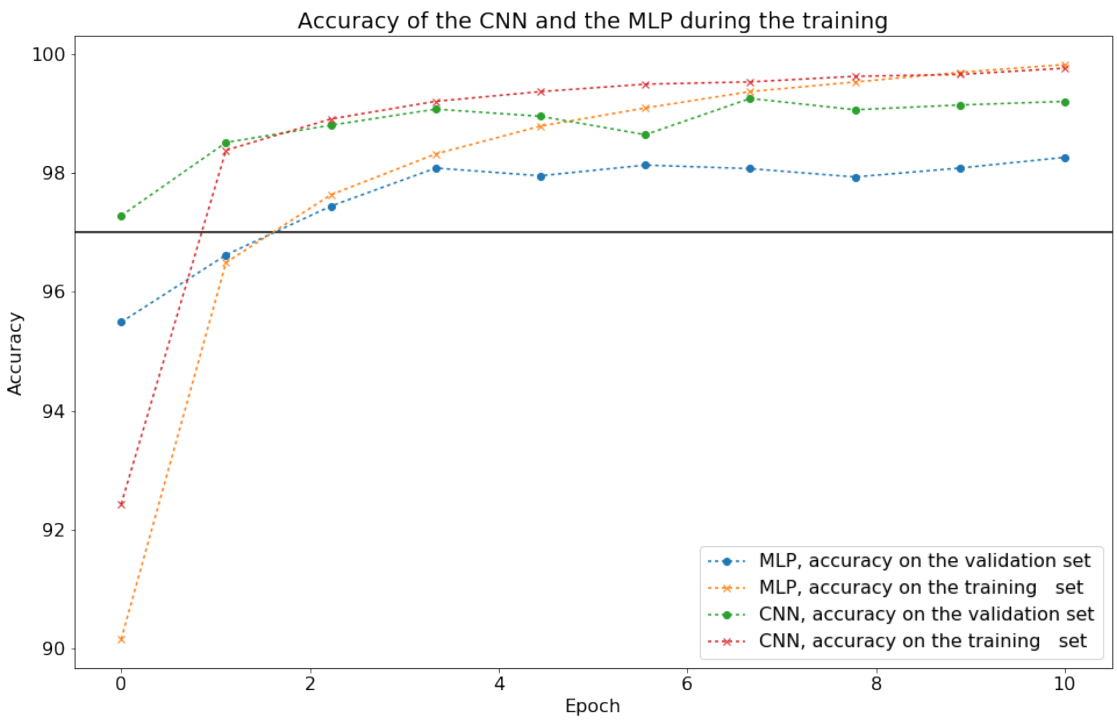
```
In [37]: # Here's what we add when we runned the notebook with small batch size:
         loading_path = "./output/2019February16_PM09H33M.png"
         IPython.display.display(IPython.display.Image(filename=loading_path))
```



3.0.2 Compare the performance of the mlp and the cnn on both dataset

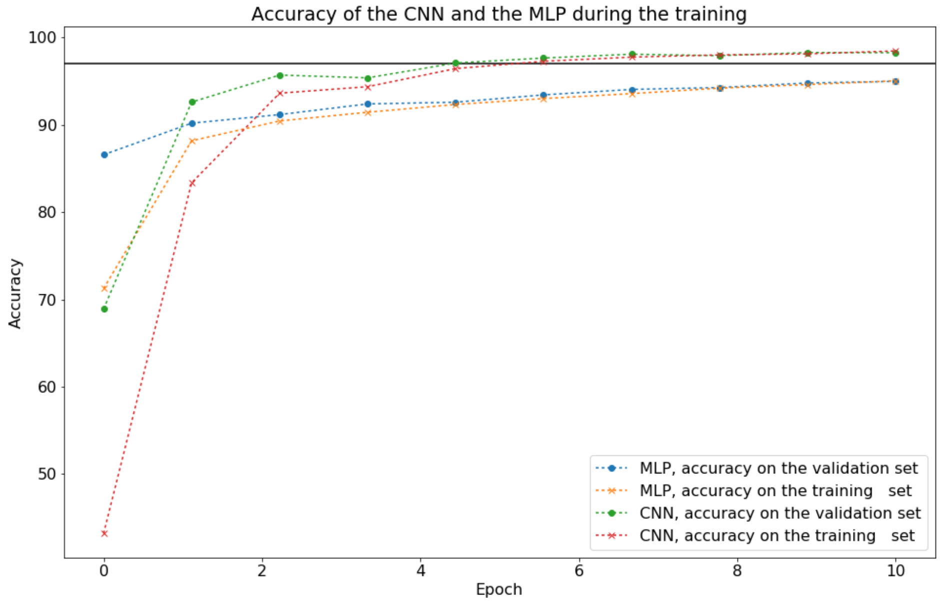
```
In [35]: # nb_epoch =  cnn_accuracy.size()[0]
         nb_epoch = 10
         x = np.linspace(0, nb_epoch, nb_epoch)
         y1a = copy.deepcopy(mlp_accuracy[:,1]).cpu().numpy()
         y1b = copy.deepcopy(mlp_accuracy[:,0]).cpu().numpy()
         y2a = copy.deepcopy(cnn_accuracy[:,1]).cpu().numpy()
         y2b = copy.deepcopy(cnn_accuracy[:,0]).cpu().numpy()

         fig, ax = plt.subplots()

         line1a_label = "MLP, accuracy on the validation set"
         line1b_label = "MLP, accuracy on the training   set"
         line2a_label = "CNN, accuracy on the validation set"
         line2b_label = "CNN, accuracy on the training   set"

         plt.axhline(y=97,color="black")

         line1a, = ax.plot(x, y1a, "o-", label=line1a_label)
         line1a.set_dashes([2, 2])  # 2pt line, 2pt break

         line1b, = ax.plot(x, y1b, "x-", label=line1b_label)
         line1b.set_dashes([2, 2])  # 2pt line, 2pt break

         line2a, = ax.plot(x, y2a, "o-", label=line2a_label)
```

```
line2a.set_dashes([2, 2])  # 2pt line, 2pt break

line2b, = ax.plot(x, y2b, "x-", label=line2b_label)
line2b.set_dashes([2, 2])  # 2pt line, 2pt break

str_title1 = "Accuracy of the CNN and the MLP during the training"
plt.title(str_title1)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')

ax.legend()
want_log     = True
path_to_save = "./output/"
filename     = datetime.datetime.now().strftime("%Y%B%d_%p%IH%MM")
if want_log :
        plt.savefig(path_to_save + filename + ".png")
plt.show()
```

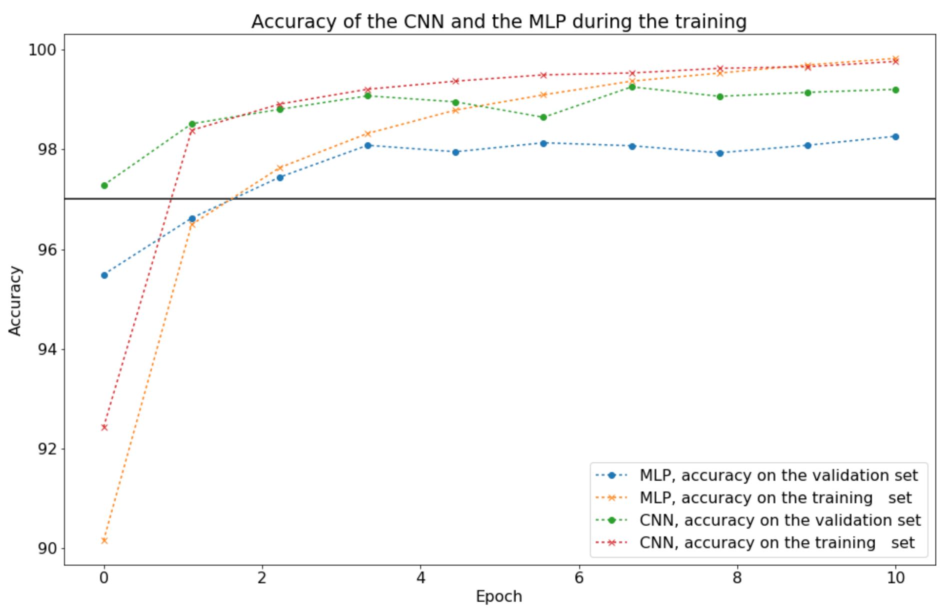


4 Comparison of accuracy

In [38]:
```
# Here's what we add when we runned the notebook (with large batch size):
loading_path = "./output/2019February15_PM12H52M.png"
IPython.display.display(IPython.display.Image(filename=loading_path))
```




```
In [36]: # Here's what we add when we runned the notebook (with small batch size):
         loading_path = "./output/2019February16_PM09H34M.png"
         IPython.display.display(IPython.display.Image(filename=loading_path))
```



4.1 Comparing the models

k-best Error : having the right label in the net's top k answers count as a good answer We measure errors

```
In [51]: batch_size = 32*64
         valid_loader = DataLoader(valid_set, batch_size=batch_size,shuffle=True, num_workers=2)
         net_list = [net_cnn,net_best_mlp]
         acc      = np.empty(2)

         for n,net in enumerate(net_list,0) :
             k = 1
             correct = torch.tensor([0], device=device)
             total   = torch.tensor([0], device=device)
             with torch.no_grad():
                 for data in valid_loader:
                     images, labels = data
                     images, labels = images.to(device), labels.to(device)
                     outputs = net(images)
                     predicted = torch.topk(outputs.data, k)[1]
                     total += labels.size(0)
                     for i in range(k) :
                         correct += (predicted[:,i] == labels).sum()
                 acc[n] = 100 - (100 * correct.double() / total.double()).item()
                 l_min = min( len(net_cnn.__doc__), 20)
             print('{name} \n{k}-best Error of the network on the {nb} test images: {acc:.3f}
        %'.format(
                         name= net.__doc__[:l_min],
                         k   = k,
                         nb  = valid_dataset_size,
                         acc = acc[n]
                  ))
         print("\n")
         print("Relative difference = {diff:.2f} %".format( diff=100*abs(acc[0]-acc[1])/acc[0] )
         )
```

```
    CNNClassifier :
1-best Error of the network on the 10000 test images: 0.800 %

    MLP with 2 hidd
1-best Error of the network on the 10000 test images: 1.740 %


Relative difference = 117.50 %
```

With small batch size : The measure of k-best error size gives us :
- a relative difference of 117.50 % for k=1
- a relative difference of 263.64 % for k=2
- a relative difference of 700.00 % for k=3
- a relative difference of inf % for k=4

Which means that the cnn has better 1st, 2nd, 3rd and 4th options than the mlp (and is making no mistakes with k=4).

With large batch size : The measure of k-best error size gives us :
- a relative difference of 189.60 % for k=1
- a relative difference of 443.75 % for k=2
- a relative difference of 492.31 % for k=3

- a relative difference of 1750.0 % for k=4
- a relative difference of inf % for k=5

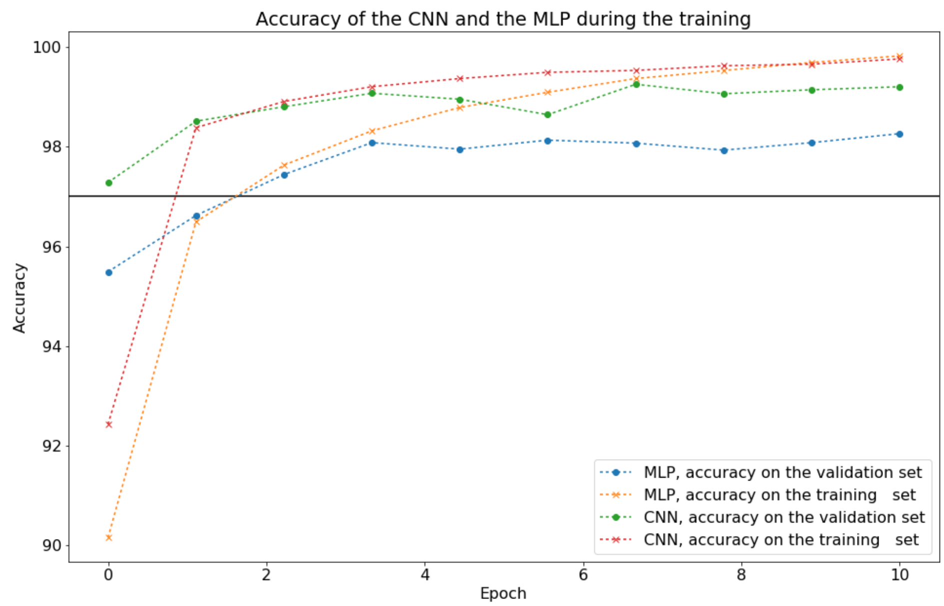
Which means that the cnn has better 1st, 2nd, 3rd, 4th and 5th options than the mlp (and is making no mistakes with k=5).

Compare the performances of CNN vs MLP. Comment The plot comparing the accuracy of the cnn and the mlp across epoch (see below) show that using a good combinaison of hyper-parameters enable both models to achieve an high accuracy on the training set (99.82 % for the mlp vs 99.76 % for the cnn) which means that the models (which share approximatively the same number of parameters (~0.88 millions params)) have enough capacity to fit the training data. The plot also shows that the mlp achieve lower accuracy on the validation set (98.26 %) than the cnn (99.20 %) and has a bigger gap between its accuracy on the two datasets. Comparing the error on the validation set of the two models after 10 epochs shows that the relative difference is >100% and that the cnn's 2nd, 3rd, 4th,... best choice are also better by a good margin that those of the mlp.

As seen during the lectures (and in the manual chap 9.4), we know that the use of convolution in a layer can be seen as an infinitely strong prior saying that the layer is constrained to learn a function that is both equivariant to translation and only contain local interractions. Max pooling can also be seen as an infinitely strong prior saying that this function should locally be invariant to small translation. These priors are reasonnables and well fitted for inputs that are images. Because they are priors, they can cause underfitting. The mlp is not making any of these assumption and we can see in our experiment that the mlp actually achieves (slightly) better accuracy on the training set, which is interresting. It indicates that these assumptions correspond to having some preferences in the set of learnable functions that are well suited for the task at hand and help to generalize.

There is also another thing that could be an important factor. In the manual chap 6.4, figure 6.6 shows how test accuracy increases with network's depth for the task of multidigits number recognition. In our case, the cnn is deeper than the mlp and the depth difference could be another factor infuencing their generalization. In our case, we were constrain to use a 2-hidden layer mlp, but it would be interresting to include a deep fully connected feed forward network in the comparison.

In [55]:
```
loading_path = "./output/2019February16_PM09H34M.png"
IPython.display.display(IPython.display.Image(filename=loading_path))
```



5 The interesting part of the notebook ends here

5.0.1 Save and load models

With the following code, you can load the models we obtained when we runned the notebook and save your models.

```
In [56]: want_to_save = False
         want_to_load = False

In [57]: local_path = "./save/export/"
         if want_to_save :
             # save current mlp states only
             mynet = copy.deepcopy(net_best_mlp).cpu()
             saving_name = "net_best_mlp_low_batch_size.pth"
             state_dict_to_disk = mynet.state_dict()
             torch.save( state_dict_to_disk , local_path + saving_name)

             # save current cnn states only
             mynet = copy.deepcopy(net_cnn).cpu()
             saving_name = "net_cnn_low_batch_size.pth"
             state_dict_to_disk = mynet.state_dict()
             torch.save( state_dict_to_disk , local_path + saving_name)

In [45]: local_path = "./save/export/"
         if want_to_load :
             # load mlp final state only
             loading_name = "net_best_mlp_low_batch_size.pth" # MLP()
             mynet = MLP()
             mynet.load_state_dict(torch.load(local_path + loading_name))
             net_best_mlp = copy.deepcopy( mynet ).to(device)
```

```python
# load cnn final state only
loading_name = "net_cnn_low_batch_size.pth" # MLP()
mynet = CNNClassifier()
mynet.load_state_dict(torch.load(local_path + loading_name))
net_cnn = copy.deepcopy( mynet ).to(device)
```