PA1_na21b033

September 10, 2023

#1. Self Implementation

##1.1. Installing necesarry packages

[38]:
```
!pip install -q torch_snippets
!pip install -q torchinfo
!pip install -q torchmetrics

from torchmetrics import ConfusionMatrix
from torch_snippets import*
from torchinfo import summary
import torch
from torchvision import datasets, transforms
from torch import nn
import matplotlib.pyplot as plt
import sklearn
from sklearn.metrics import classification_report
import pandas as pd
```

[39]:
```
device="cuda" if torch.cuda.is_available() else "cpu"
device #device agniostic code
```

[39]: 'cuda'

##1.2. Getting the data

[40]:
```
transformer= transforms.Compose([transforms.ToTensor()]) #dataset contains imgs␣
↪in PIL format
```

[41]:
```
train_ds=datasets.MNIST(root="MNIST/", train=True, download=True, transform=␣
↪transformer, target_transform=torchvision.transforms.Compose([lambda x:torch.
↪LongTensor([x]), lambda x:F.one_hot(x,10)]))
#convert labels to one hot encoding
test_ds=datasets.MNIST(root="MNIST/", train=False, download=True, transform=␣
↪transformer, target_transform=torchvision.transforms.Compose([lambda x:torch.
↪LongTensor([x]), lambda x:F.one_hot(x,10)]))

train_dl= torch.utils.data.DataLoader(train_ds, batch_size=64, drop_last=True)
test_dl= torch.utils.data.DataLoader(test_ds, batch_size=64, drop_last=True)
```

[42]:
```
im, label=next(iter(train_dl))
im.shape, label.shape #labels are not one hot encoded; img_shape-> [28*28]
```

[42]:
```
(torch.Size([64, 1, 28, 28]), torch.Size([64, 1, 10]))
```

[43]:
```
im=im.view([64,-1])
im.shape, label.shape
```

[43]:
```
(torch.Size([64, 784]), torch.Size([64, 1, 10]))
```

##1.3. Building the model

[44]:
```
class Layer:
    def __init__(self):
        self.input = None
        self.output = None

    def forward_propagation(self, input):
        raise NotImplementedError

    def backward_propagation(self, output_error, learning_rate):
        raise NotImplementedError
```

[45]:
```
class Linear_l(Layer):
  def __init__(self, in_features, out_features):
    self.m1=(6/(in_features+out_features))**0.5
    self.weights= torch.FloatTensor(in_features, out_features).uniform_(-self.
↪m1, self.m1).to(device) #glorot initialization
    self.bias= torch.zeros(1, out_features).to(device)

  def forward_prop(self, input_data):
    self.input=input_data
    self.output= torch.matmul(self.input, self.weights)+self.bias
    return self.output

  def back_prop(self, output_error, lr):
    weights_error= torch.matmul(self.input.T, output_error)
    self.weights= self.weights-lr*weights_error
    self.bias= self.bias-lr*output_error
    input_error= torch.matmul(output_error, self.weights.T)
    return input_error
```

[46]:
```
class Activation_l(Layer):
  def __init__(self, activation, activation_g): #activation_g-> activation␣
↪gradient
    self.activation=activation
    self.activation_g=activation_g
```

```
  def forward_prop(self, input_data):
    self.input=input_data
    return self.activation(self.input)

  def back_prop(self, output_error, lr):
    return torch.mul(self.activation_g(self.input), output_error)
```

[47]:
```
'''Activation function ->g indicates gradient || derivative of the function'''

def relu(x):
  return nn.functional.relu(x)

def relu_g(x):
  x1=x.clone()
  x1[x<=0]=0
  x1[x>0]=1
  return x1

def tanh(x):
  return nn.functional.tanh(x)

def tanh_g(x):
  return 1-(tanh(x))**2

def softmax(x):
  return nn.functional.softmax(x, dim=-1)

def softmax_g(x):
  output= softmax(x)*(1-softmax(x))
  return output

''' From experimentation it has been found that relu serves as the better␣
↪activation this is because of the vanishing gradients issue of the tanh␣
↪activation function'''
```

[47]:
```
' From experimentation it has been found that relu serves as the better
activation this is because of the vanishing gradients issue of the tanh
activation function'
```

[48]:
```
def accuracy(y_t, y_p):
  y_t, y_p= torch.argmax(y_t, dim=-1), torch.argmax(y_p, dim=-1)
  return torch.sum(torch.eq(y_t, y_p))/len(y_t)
```

[49]:
```
''' Loss function and its gradient'''

def cross_entropy(y_t, y_p):
```

```
  output=-torch.log(torch.sum(torch.mul(y_p, y_t), dim=1))
  return torch.sum(output)/len(y_t)

def cross_entropy_g(y_t, y_p):
  return -2*(y_t-y_p)/len(y_t)
```

[50]:
```
class network:
  def __init__(self):
    self.layers=[]
    self.loss=None
    self.loss_g=None

  def add(self, layer):
    self.layers.append(layer)

  def use(self, loss, loss_g):
    self.loss= loss
    self.loss_g= loss_g

  def train_step(self, data, lr):
    x_t, y_t= data
    x_t, y_t=x_t.view([64,-1]).to(device), y_t.to(device).squeeze()
    output= x_t
    for layer in self.layers:
      output= layer.forward_prop(output)
    err=self.loss(y_t, output)
    error=self.loss_g(y_t, output)
    for layer in reversed(self.layers):
      error= layer.back_prop(error, lr)
    train_acc=accuracy(y_t, output)
    return err, train_acc

  def test_step(self, data):
    x, y= data
    x, y= x.view([64, -1]).to(device), y.to(device).squeeze()
    output= x
    for layer in self.layers:
      output= layer.forward_prop(output)
    err=self.loss(y, output)
    test_acc=accuracy(y, output)
    return err, test_acc, output
```

[51]:
```
net=network()
net.add(Linear_l(28*28, 500))
net.add(Activation_l(relu, relu_g))
net.add(Linear_l(500, 250))
net.add(Activation_l(relu, relu_g))
```

```
net.add(Linear_l(250, 100))
net.add(Activation_l(relu, relu_g))
net.add(Linear_l(100, 10))
net.add(Activation_l(softmax, softmax_g))

net.use(cross_entropy, cross_entropy_g)
```

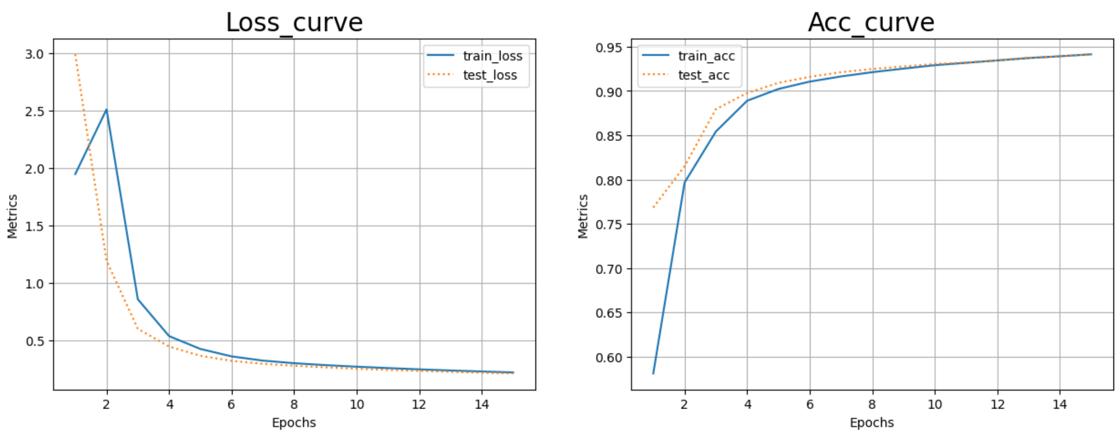
[52]:
```
EPOCHS=15
log=Report(EPOCHS)
for epoch in range(EPOCHS):
  n=len(train_dl)
  for idx, data in enumerate(train_dl):
    loss, t_acc= net.train_step(data, lr=0.01)
    log.record((epoch+(1+idx)/n), train_loss= loss, train_acc= t_acc, end="\r")

  n=len(test_dl)
  for idx, data in enumerate(test_dl):
    loss, t_acc, output= net.test_step(data)
    log.record((epoch+(1+idx)/n), test_loss= loss, test_acc= t_acc, end="\r")
  log.report_avgs(epoch+1)
```

```
EPOCH: 1.000  test_loss: 2.986  test_acc: 0.768  train_acc: 0.581  train_loss:
1.946  (15.42s - 215.81s remaining)
EPOCH: 2.000  test_loss: 1.193  test_acc: 0.815  train_acc: 0.797  train_loss:
2.508  (36.60s - 237.89s remaining)
EPOCH: 3.000  test_loss: 0.604  test_acc: 0.879  train_acc: 0.854  train_loss:
0.860  (48.26s - 193.03s remaining)
EPOCH: 4.000  test_loss: 0.449  test_acc: 0.898  train_acc: 0.889  train_loss:
0.540  (60.28s - 165.76s remaining)
EPOCH: 5.000  test_loss: 0.369  test_acc: 0.909  train_acc: 0.902  train_loss:
0.429  (72.14s - 144.28s remaining)
EPOCH: 6.000  test_loss: 0.325  test_acc: 0.916  train_acc: 0.911  train_loss:
0.364  (84.49s - 126.73s remaining)
EPOCH: 7.000  test_loss: 0.299  test_acc: 0.921  train_acc: 0.916  train_loss:
0.327  (96.97s - 110.82s remaining)
EPOCH: 8.000  test_loss: 0.282  test_acc: 0.925  train_acc: 0.921  train_loss:
0.305  (110.00s - 96.25s remaining)
EPOCH: 9.000  test_loss: 0.268  test_acc: 0.928  train_acc: 0.925  train_loss:
0.288  (121.95s - 81.30s remaining)
EPOCH: 10.000  test_loss: 0.257  test_acc: 0.930  train_acc: 0.929  train_loss:
0.274  (133.88s - 66.94s remaining)
EPOCH: 11.000  test_loss: 0.247  test_acc: 0.932  train_acc: 0.932  train_loss:
0.262  (145.80s - 53.02s remaining)
EPOCH: 12.000  test_loss: 0.238  test_acc: 0.935  train_acc: 0.934  train_loss:
0.251  (162.35s - 40.59s remaining)
EPOCH: 13.000  test_loss: 0.229  test_acc: 0.938  train_acc: 0.937  train_loss:
0.242  (188.94s - 29.07s remaining)
```

```
EPOCH: 14.000  test_loss: 0.222  test_acc: 0.939  train_acc: 0.939  train_loss:
0.233  (200.95s - 14.35s remaining)
EPOCH: 15.000  test_loss: 0.216  test_acc: 0.941  train_acc: 0.941  train_loss:
0.225  (213.06s - 0.00s remaining)
```

[53]:
```
fig, ax= plt.subplots(ncols=2, figsize=(15,5)) #w,h
log.plot_epochs(["train_loss", "test_loss"], ax=ax[0], title="Loss_curve");␣
↪#loss_curve
ax[0].legend(loc='upper right', fontsize=10)
log.plot_epochs(["train_acc", "test_acc"], ax=ax[1], title="Acc_curve");␣
↪#acc_curve
```

```
100%|      | 116/116 [00:00<00:00, 300.55it/s]
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note
that artists whose label start with an underscore are ignored when legend() is
called with no argument.
100%|      | 116/116 [00:00<00:00, 297.32it/s]
```

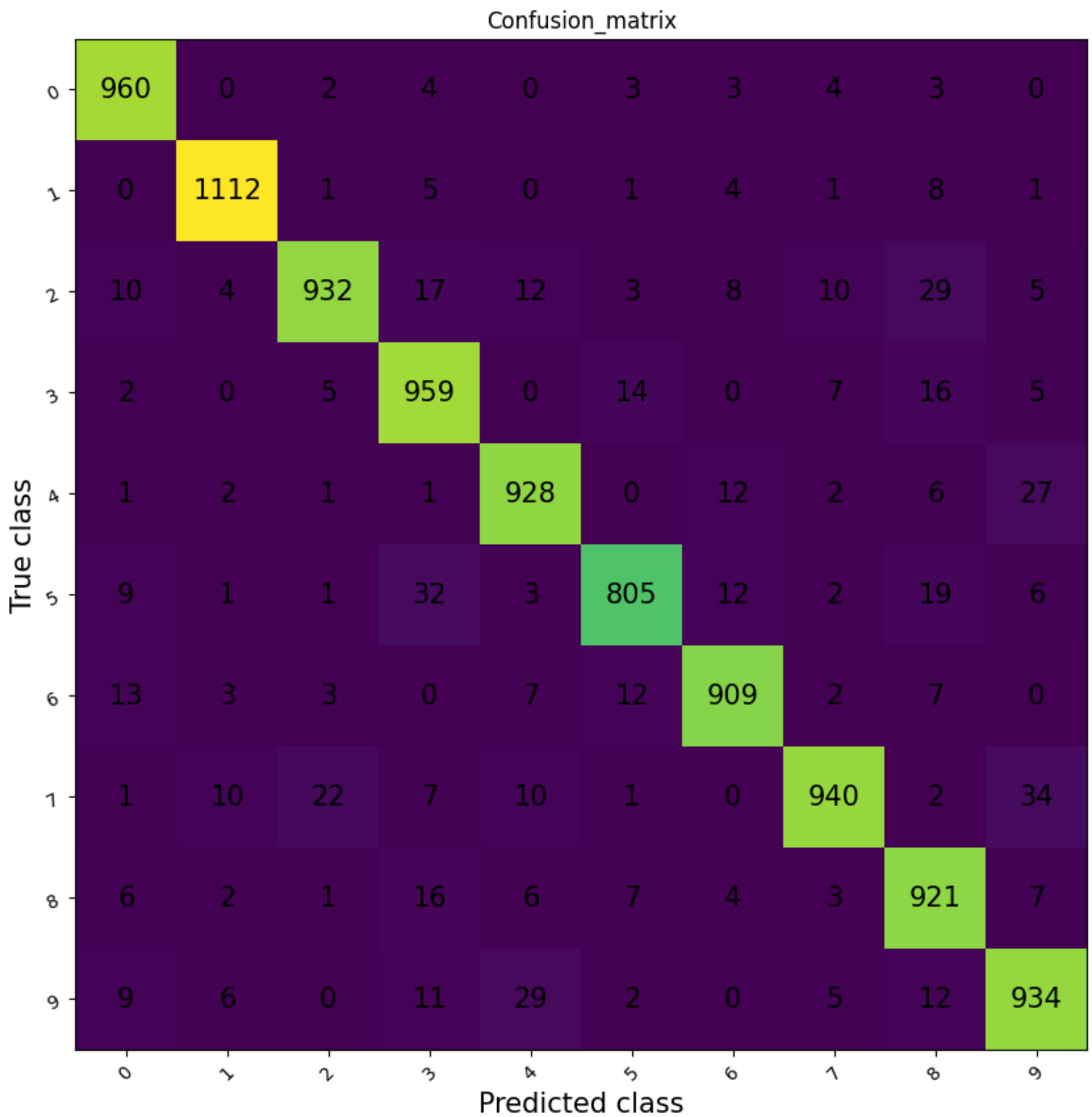


##1.4. Confusion matrix

[54]:
```
cm=ConfusionMatrix(task="multiclass", num_classes=10)
```

[55]:
```
labels, result=[], []
for data in test_dl:
  _, label=data
  l, a, outputs= net.test_step(data)
  result.append(outputs.argmax(dim=-1))
  labels.append(label.argmax(dim=-1))
cm(torch.flatten(torch.stack(result).to("cpu")), torch.flatten(torch.
↪stack(labels).to("cpu")));
```

[56]:
```
fig, ax = plt.subplots(figsize=(10, 10))
cm.plot(ax=ax)
ax.set_title("Confusion_matrix");
```



[ ]:
```
'''Our model is performing well, for an ideal confusion matrix all elements␣
↪along the diagonal shd be maximum as it means that the true labels were␣
↪identified correctly'''
```

##1.5. Classification Report

[58]:
```
c_report_1=classification_report(torch.flatten(torch.stack(labels).to("cpu")),␣
↪torch.flatten(torch.stack(result).to("cpu")), output_dict=True)
```

```
df_1= pd.DataFrame(c_report_1)
df_1
```

[58]:
```
                  0            1            2            3           4  \
precision    0.949555     0.975439     0.962810     0.911597    0.932663
recall       0.980592     0.981465     0.904854     0.951389    0.946939
f1-score     0.964824     0.978443     0.932933     0.931068    0.939747
support    979.000000  1133.000000  1030.000000  1008.000000  980.000000

                  5           6            7           8            9  \
precision    0.949292    0.954832     0.963115    0.900293     0.916585
recall       0.904494    0.950837     0.915287    0.946557     0.926587
f1-score     0.926352    0.952830     0.938592    0.922846     0.921559
support    890.000000  956.000000  1027.000000  973.000000  1008.000000

           accuracy    macro avg  weighted avg
precision  0.941506     0.941618      0.942117
recall     0.941506     0.940900      0.941506
f1-score   0.941506     0.940919      0.941474
support    0.941506  9984.000000   9984.000000
```

2. Using PyTorch Libraries

2.1. Building the model

[59]:
```
model= nn.Sequential(nn.Flatten(),
                     nn.Linear(28*28, 500), nn.ReLU(),
                     nn.Linear(500, 250), nn.ReLU(),
                     nn.Linear(250, 100), nn.ReLU(),
                     nn.Linear(100, 10), nn.Softmax(dim=1)).to(device)
```

[60]:
```
summary(model, input_size=(64, 28*28))
```

[60]:
```
================================================================================
==========
Layer (type:depth-idx)                   Output Shape              Param #
================================================================================
==========
Sequential                                [64, 10]                  --
 Flatten: 1-1                            [64, 784]                 --
 Linear: 1-2                             [64, 500]                 392,500
 ReLU: 1-3                               [64, 500]                 --
 Linear: 1-4                             [64, 250]                 125,250
 ReLU: 1-5                               [64, 250]                 --
 Linear: 1-6                             [64, 100]                 25,100
 ReLU: 1-7                               [64, 100]                 --
 Linear: 1-8                             [64, 10]                  1,010
 Softmax: 1-9                            [64, 10]                  --
```

```
================================================================================
==========
Total params: 543,860
Trainable params: 543,860
Non-trainable params: 0
Total mult-adds (M): 34.81
================================================================================
==========
Input size (MB): 0.20
Forward/backward pass size (MB): 0.44
Params size (MB): 2.18
Estimated Total Size (MB): 2.82
================================================================================
==========
```

##2.2. Run the model

[61]:
```
img, l= next(iter(train_dl))
```

[62]:
```
img.shape, l.shape
```

[62]:
```
(torch.Size([64, 1, 28, 28]), torch.Size([64, 1, 10]))
```

[63]:
```
optimizer=torch.optim.SGD(params= model.parameters(), lr=0.01) #torch.optim.
 ↪Adam(params= model.parameters()) #Adam converges faster
criterion= cross_entropy
EPOCHS=15
```

[64]:
```
def train_step(model, data, criterion, optimizer):
  model.train()
  img, label= data
  img, label= img.to(device), label.to(device).squeeze()
  output= model(img)
  loss= criterion(label, output)
  t_acc= accuracy(label, output)
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
  return loss.item(), t_acc

def test_step(model, data, criterion):
  model.eval()
  with torch.inference_mode():
    img, label= data
    img, label= img.to(device), label.to(device).squeeze()
    output= model(img)
    loss= criterion(label, output)
```

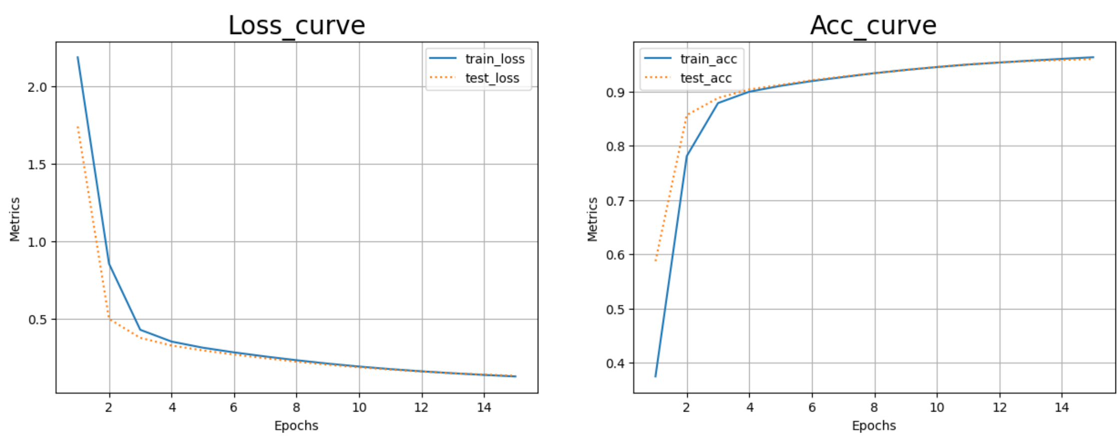
```
    t_acc= accuracy(label, output)
    return loss.item(), t_acc
```

[65]:
```
log=Report(EPOCHS)
for epoch in range(EPOCHS):
  n=len(train_dl)
  for idx, data in enumerate(train_dl):
    loss, t_acc= train_step(model, data, criterion, optimizer)
    log.record((epoch+(1+idx)/n), train_loss= loss, train_acc= t_acc, end="\r")

  n=len(test_dl)
  for idx, data in enumerate(test_dl):
    loss, t_acc= test_step(model, data, criterion)
    log.record((epoch+(1+idx)/n), test_loss= loss, test_acc= t_acc, end="\r")
  log.report_avgs(epoch+1)
```

```
EPOCH: 1.000  test_loss: 1.739  test_acc: 0.587  train_acc: 0.375  train_loss:
2.185  (30.98s - 433.70s remaining)
EPOCH: 2.000  test_loss: 0.502  test_acc: 0.856  train_acc: 0.781  train_loss:
0.857  (47.11s - 306.18s remaining)
EPOCH: 3.000  test_loss: 0.379  test_acc: 0.888  train_acc: 0.878  train_loss:
0.431  (65.07s - 260.26s remaining)
EPOCH: 4.000  test_loss: 0.330  test_acc: 0.903  train_acc: 0.899  train_loss:
0.356  (81.45s - 223.99s remaining)
EPOCH: 5.000  test_loss: 0.298  test_acc: 0.912  train_acc: 0.910  train_loss:
0.316  (97.29s - 194.58s remaining)
EPOCH: 6.000  test_loss: 0.272  test_acc: 0.920  train_acc: 0.919  train_loss:
0.286  (115.03s - 172.54s remaining)
EPOCH: 7.000  test_loss: 0.248  test_acc: 0.927  train_acc: 0.926  train_loss:
0.260  (131.65s - 150.45s remaining)
EPOCH: 8.000  test_loss: 0.226  test_acc: 0.933  train_acc: 0.933  train_loss:
0.235  (145.55s - 127.35s remaining)
EPOCH: 9.000  test_loss: 0.207  test_acc: 0.939  train_acc: 0.939  train_loss:
0.214  (159.50s - 106.33s remaining)
EPOCH: 10.000  test_loss: 0.190  test_acc: 0.945  train_acc: 0.945  train_loss:
0.195  (171.83s - 85.91s remaining)
EPOCH: 11.000  test_loss: 0.175  test_acc: 0.950  train_acc: 0.949  train_loss:
0.178  (183.91s - 66.88s remaining)
EPOCH: 12.000  test_loss: 0.163  test_acc: 0.953  train_acc: 0.953  train_loss:
0.164  (196.25s - 49.06s remaining)
EPOCH: 13.000  test_loss: 0.152  test_acc: 0.956  train_acc: 0.956  train_loss:
0.152  (208.56s - 32.09s remaining)
EPOCH: 14.000  test_loss: 0.143  test_acc: 0.958  train_acc: 0.960  train_loss:
0.141  (220.67s - 15.76s remaining)
EPOCH: 15.000  test_loss: 0.136  test_acc: 0.959  train_acc: 0.963  train_loss:
0.131  (232.98s - 0.00s remaining)
```

[66]:
```
fig, ax= plt.subplots(ncols=2, figsize=(15,5)) #w,h
log.plot_epochs(["train_loss", "test_loss"], ax=ax[0], title="Loss_curve"); #loss_curve
ax[0].legend(loc='upper right', fontsize=10)
log.plot_epochs(["train_acc", "test_acc"], ax=ax[1], title="Acc_curve"); #acc_curve
```

```
100%|          | 116/116 [00:00<00:00, 301.38it/s]
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note
that artists whose label start with an underscore are ignored when legend() is
called with no argument.
100%|          | 116/116 [00:00<00:00, 299.50it/s]
```

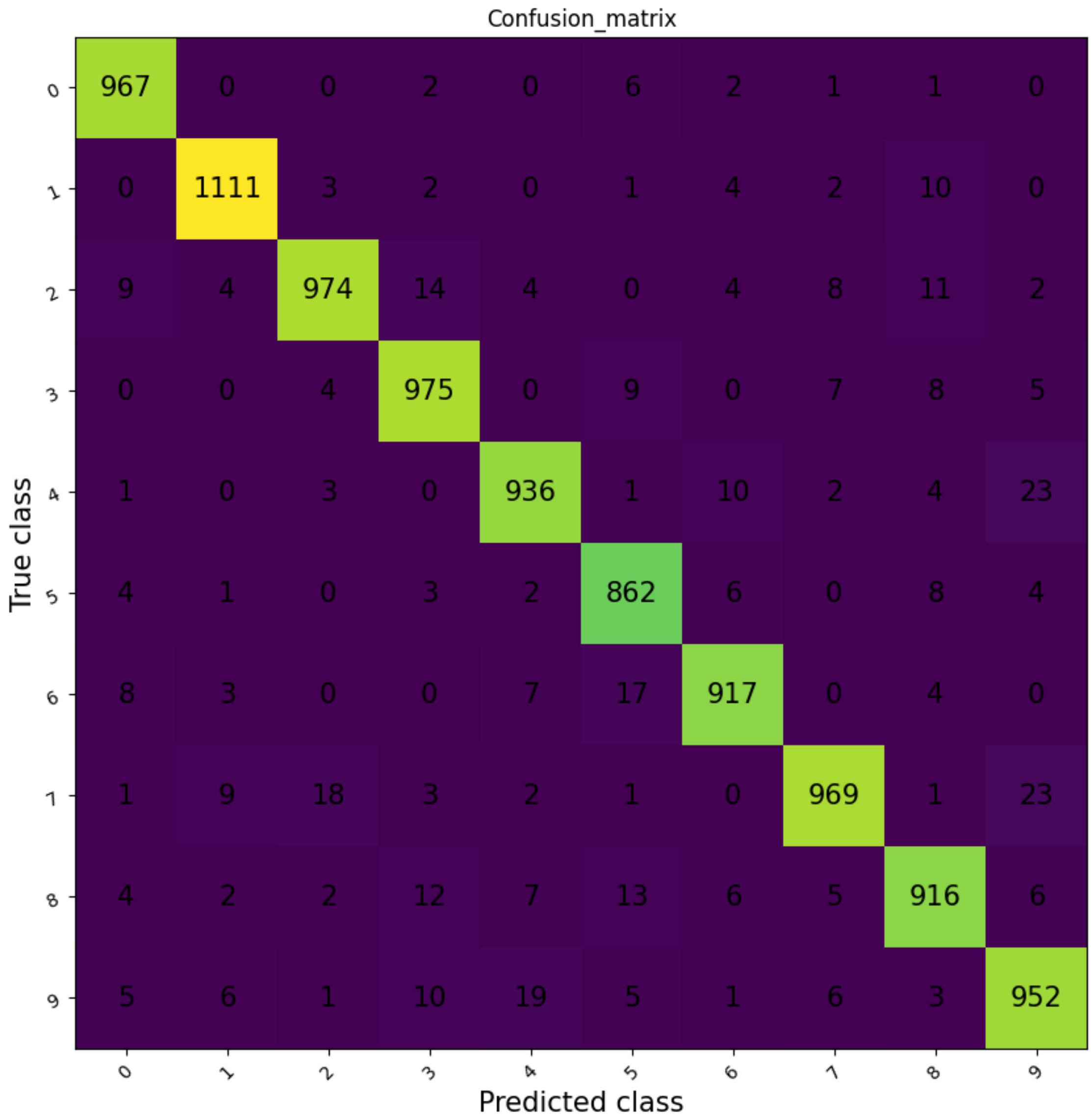


##2.3. Confusion matrix

[67]:
```
cm=ConfusionMatrix(task="multiclass", num_classes=10)
```

[68]:
```
labels, output=[], []

model.eval()
with torch.inference_mode():
  for data in test_dl:
    im, label=data
    output.append(model(im.to(device)).argmax(dim=-1))
    labels.append(label.argmax(dim=-1))
cm(torch.flatten(torch.stack(output)).to("cpu"), torch.flatten(torch.stack(labels)).to("cpu"));
```

[69]:
```
fig, ax = plt.subplots(figsize=(10, 10))
cm.plot(ax=ax)
ax.set_title("Confusion_matrix");
```



[ ]:
```
'''Our model is performing well, for an ideal confusion matrix all elements␣
↪along the diagonal shd be maximum as it means that the true labels were␣
↪identified correctly'''
```

##2.4. Classification_report

[70]:
```
c_report_2=classification_report(torch.flatten(torch.stack(labels).to("cpu")),␣
↪torch.flatten(torch.stack(result).to("cpu")), output_dict=True)
df_2=pd.DataFrame(c_report_2)
df_2
```

[70]:

```
                    0            1            2            3           4  \
precision    0.949555     0.975439     0.962810     0.911597    0.932663
recall       0.980592     0.981465     0.904854     0.951389    0.946939
f1-score     0.964824     0.978443     0.932933     0.931068    0.939747
support    979.000000  1133.000000  1030.000000  1008.000000  980.000000

                    5           6            7           8            9  \
precision    0.949292    0.954832     0.963115    0.900293     0.916585
recall       0.904494    0.950837     0.915287    0.946557     0.926587
f1-score     0.926352    0.952830     0.938592    0.922846     0.921559
support    890.000000  956.000000  1027.000000  973.000000  1008.000000

           accuracy    macro avg  weighted avg
precision  0.941506     0.941618      0.942117
recall     0.941506     0.940900      0.941506
f1-score   0.941506     0.940919      0.941474
support    0.941506  9984.000000   9984.000000
```

[70]: