ypdfabc9s

March 12, 2024

[2]:
```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.nn.init as init

import matplotlib.pyplot as plt
import numpy as np

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

cpu

Model

[3]:
```
class MLP_CIFR10(nn.Module):
  # Batch norm = True enables the Batch-Normalization
  def __init__(self, input_size=3*32*32, h1_size = 500, h2_size = 250, h3_size␣
↪= 100, num_classes = 10, batch_norm = False):
    super(MLP_CIFR10, self).__init__();
    self.batch_norm = batch_norm
    # Hidden Layer 1 with ReLU activation
    self.fc1 = nn.Linear(in_features = input_size, out_features = h1_size);
    if batch_norm == True:
      # print("Batch norm1 enabled")
      self.bn1 = nn.BatchNorm1d(h1_size)
    self.relu1 = nn.ReLU()
    # Hidden Layer 2 with ReLU activation
    self.fc2 = nn.Linear(in_features = h1_size, out_features = h2_size);
    if batch_norm == True:
      # print("Batch norm2 enabled")
      self.bn2 = nn.BatchNorm1d(h2_size)
    self.relu2 = nn.ReLU()
    # Hidden Layer 3 with ReLU activation
    self.fc3 = nn.Linear(in_features = h2_size, out_features = h3_size);
```

```
        if batch_norm == True:
          # print("Batch norm3 enabled")
          self.bn3 = nn.BatchNorm1d(h3_size)
        self.relu3 = nn.ReLU()
        # output layer
        self.fc_out = nn.Linear(in_features = h3_size, out_features = num_classes);
        self.softmax = nn.Softmax(dim = 1); # we need the output to be y0 to y9␣
↪depicitng the classes

    def forward(self, x):
        x = torch.flatten(x, 1); # flatten all dimensions except 1st dimension␣
↪which represents batch size

        # Generate the output of Fully connected layer 1
        out = self.fc1(x);
        if self.batch_norm == True:
          out = self.bn1(out)
        out = self.relu1(out);

        # Generate the output of Fully connected layer 2
        out = self.fc2(out);
        if self.batch_norm == True:
          out = self.bn2(out)
        out = self.relu2(out);

        # Generate the output of Fully connected layer 3
        out = self.fc3(out);
        if self.batch_norm == True:
          out = self.bn3(out)
        out = self.relu3(out);

        # Final output layer
        out = self.fc_out(out);
        # softmax has not been applied here since the nn.CrossEntropyLoss() applies␣
↪LogSoftmax on an input, followed by NLLoss
        # Refernece for the above statemnet : https://pytorch.org/docs/stable/
↪generated/torch.nn.CrossEntropyLoss.html#torch.nn.CrossEntropyLoss
        return out
```

Initialize the model

[4]:
```
model = MLP_CIFR10(batch_norm=True).to(device) # Batch Normalization Turned off
```

Creating Data Loader

[5]:
```
# Normalize input data to 0.5 mean and 0.5 SD
```

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

# load Training Dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

#Load Testing Dataset
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

# Different classes present in the data set
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./data/cifar-10-python.tar.gz

100%|     | 170498071/170498071 [00:02<00:00, 57925483.89it/s]

Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

Define Loss Function and Optimizer

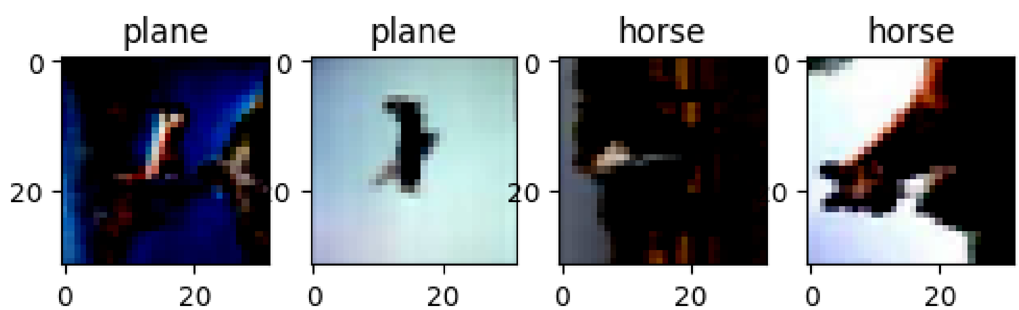
[6]:
```
# Cross entropy loss
criterion = nn.CrossEntropyLoss()
# Stochastic Gradient Descent Optimiser, with learning rate = 0.001 and␣
↪Momentum = 0.9
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

Cross verify the Images with Labels

[7]:
```
dataiter = iter(trainloader)

image, label = next(dataiter); #image size :  torch.Size([4, 3, 32, 32]), Batch␣
↪size = 4

for i in range(batch_size):
  image_to_show = image.T[:,:,:,i] # number of channels, Height of image, width␣
↪of image, batch size
  plt.subplot(1,batch_size, i+1)
```

```
  plt.imshow(image_to_show)
  plt.title(classes[label[i]])
```

```
<ipython-input-7-8d0e3d1b7322>:6: UserWarning: The use of `x.T` on tensors of
dimension other than 2 to reverse their shape is deprecated and it will throw an
error in a future release. Consider `x.mT` to transpose batches of matrices or
`x.permute(*torch.arange(x.ndim - 1, -1, -1))` to reverse the dimensions of a
tensor. (Triggered internally at ../aten/src/ATen/native/TensorShape.cpp:3614.)
  image_to_show = image.T[:,:,:,i] # number of channels, Height of image, width
of image, batch size
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```



Train the Model

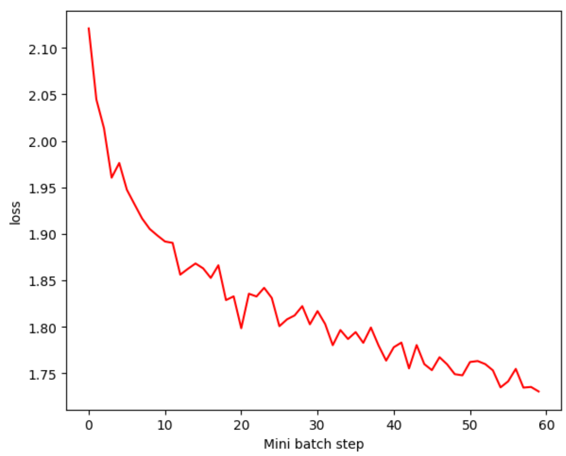
[8]:
```
# start training the model
training_error = [] # TO caopture the training error
interm_training_acc = [] # to capture intermediate training accuracy
interm_total = 0
interm_correct = 0
total = 0      # Track the total images
correct = 0    # Track the number of matched labels for images

for epoch in range(10):

  running_loss = 0.0;

  for i, data in enumerate(trainloader):
    inputs, labels = data
```

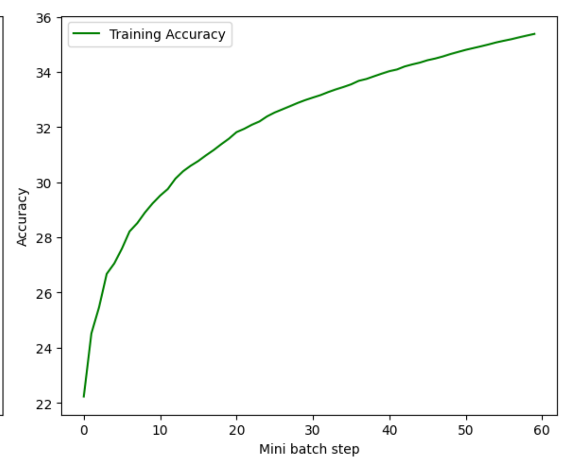
```
    inputs = inputs.to(device)
    labels = labels.to(device)

    # reset the gradients to zero
    optimizer.zero_grad()

    # forward pass
    outputs = model(inputs)

     # Loss calculation with respect to Ground Truth labels
    loss = criterion(outputs, labels)

    # Back prop
    loss.backward()

    # Gradient update
    optimizer.step()

    # cumulative loss
    running_loss += loss.item()

    predicted_probability, predicted = torch.max(outputs.data, 1) # For each␣
↪input image 10 probablities for classification will be predicted,
                                                                # from that␣
↪we need to pick the maximum probability class as our predicted class
    total += labels.size(0)
    correct += (predicted == labels).sum().item()
    interm_total, interm_correct = total, correct
    if ((i) % 2000) == 1999:    # print every 2000 mini-batches, each mini␣
↪batch has 4 images
      avg_training_loss = running_loss / 2000;
      print(f'[Epoch : {epoch + 1}, Step : {i + 1:5d}] , Average Training loss:␣
↪{avg_training_loss}')
      training_error.append(avg_training_loss)
      interm_training_acc.append((100*correct)/total)
      running_loss = 0.0

# Plot the metrics
fig = plt.figure(figsize=(12, 5))
fig.add_subplot(1,2,1)
plt.plot(training_error,color = 'r', label='Training Loss')
plt.xlabel('Mini batch step')
plt.ylabel('loss')

fig.add_subplot(1,2,2)
plt.plot(interm_training_acc,color = 'g', label='Training Accuracy')
```

```python
plt.xlabel('Mini batch step')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

print("\nTraining Finished with accuracy : ", ((100 * correct)//total) )
```

```
[Epoch : 1, Step :  2000] , Average Training loss: 2.1208820068240164
[Epoch : 1, Step :  4000] , Average Training loss: 2.044439363479614
[Epoch : 1, Step :  6000] , Average Training loss: 2.013462982416153
[Epoch : 1, Step :  8000] , Average Training loss: 1.9603595144152641
[Epoch : 1, Step : 10000] , Average Training loss: 1.9761992495059968
[Epoch : 1, Step : 12000] , Average Training loss: 1.9473756283521653
[Epoch : 2, Step :  2000] , Average Training loss: 1.931783806592226
[Epoch : 2, Step :  4000] , Average Training loss: 1.916418526381254
[Epoch : 2, Step :  6000] , Average Training loss: 1.9051845908164977
[Epoch : 2, Step :  8000] , Average Training loss: 1.8982056157290936
[Epoch : 2, Step : 10000] , Average Training loss: 1.8917197731137276
[Epoch : 2, Step : 12000] , Average Training loss: 1.890271152704954
[Epoch : 3, Step :  2000] , Average Training loss: 1.8560891939401627
[Epoch : 3, Step :  4000] , Average Training loss: 1.8623239113986492
[Epoch : 3, Step :  6000] , Average Training loss: 1.868098176598549
[Epoch : 3, Step :  8000] , Average Training loss: 1.8628684123307466
[Epoch : 3, Step : 10000] , Average Training loss: 1.8525812787115574
[Epoch : 3, Step : 12000] , Average Training loss: 1.866191026866436
[Epoch : 4, Step :  2000] , Average Training loss: 1.8287034362852574
[Epoch : 4, Step :  4000] , Average Training loss: 1.8327840587794781
[Epoch : 4, Step :  6000] , Average Training loss: 1.7984527839124202
[Epoch : 4, Step :  8000] , Average Training loss: 1.8355871742665768
[Epoch : 4, Step : 10000] , Average Training loss: 1.8326297475397586
[Epoch : 4, Step : 12000] , Average Training loss: 1.8419099759459496
[Epoch : 5, Step :  2000] , Average Training loss: 1.831062102675438
[Epoch : 5, Step :  4000] , Average Training loss: 1.8006542679667472
[Epoch : 5, Step :  6000] , Average Training loss: 1.808024291485548
[Epoch : 5, Step :  8000] , Average Training loss: 1.8122497361600398
[Epoch : 5, Step : 10000] , Average Training loss: 1.822231886357069
[Epoch : 5, Step : 12000] , Average Training loss: 1.8026556614935398
[Epoch : 6, Step :  2000] , Average Training loss: 1.8169165388941766
[Epoch : 6, Step :  4000] , Average Training loss: 1.8030190093815326
[Epoch : 6, Step :  6000] , Average Training loss: 1.7802314735949039
[Epoch : 6, Step :  8000] , Average Training loss: 1.7965243506133557
[Epoch : 6, Step : 10000] , Average Training loss: 1.786886178612709
[Epoch : 6, Step : 12000] , Average Training loss: 1.7943109196424485
[Epoch : 7, Step :  2000] , Average Training loss: 1.782733409613371
[Epoch : 7, Step :  4000] , Average Training loss: 1.7992254087328912
```

```
[Epoch : 7, Step :  6000] , Average Training loss: 1.77999899366498
[Epoch : 7, Step :  8000] , Average Training loss: 1.7636361369788647
[Epoch : 7, Step : 10000] , Average Training loss: 1.7780227085351945
[Epoch : 7, Step : 12000] , Average Training loss: 1.7830157733559608
[Epoch : 8, Step :  2000] , Average Training loss: 1.7552678887546063
[Epoch : 8, Step :  4000] , Average Training loss: 1.780381691068411
[Epoch : 8, Step :  6000] , Average Training loss: 1.7599116161763668
[Epoch : 8, Step :  8000] , Average Training loss: 1.7533734089136124
[Epoch : 8, Step : 10000] , Average Training loss: 1.7673154369294644
[Epoch : 8, Step : 12000] , Average Training loss: 1.7596115807294845
[Epoch : 9, Step :  2000] , Average Training loss: 1.7490563966929913
[Epoch : 9, Step :  4000] , Average Training loss: 1.7477878030538558
[Epoch : 9, Step :  6000] , Average Training loss: 1.7621385582685472
[Epoch : 9, Step :  8000] , Average Training loss: 1.7631915155649185
[Epoch : 9, Step : 10000] , Average Training loss: 1.759877965092659
[Epoch : 9, Step : 12000] , Average Training loss: 1.753155266880989
[Epoch : 10, Step :  2000] , Average Training loss: 1.7348489750027656
[Epoch : 10, Step :  4000] , Average Training loss: 1.7413458691388368
[Epoch : 10, Step :  6000] , Average Training loss: 1.754765149652958
[Epoch : 10, Step :  8000] , Average Training loss: 1.734613249450922
[Epoch : 10, Step : 10000] , Average Training loss: 1.7353137172162534
[Epoch : 10, Step : 12000] , Average Training loss: 1.730467726945877
```






```
Training Finished with accuracy :  35
```

Accuracy for 10,000 images

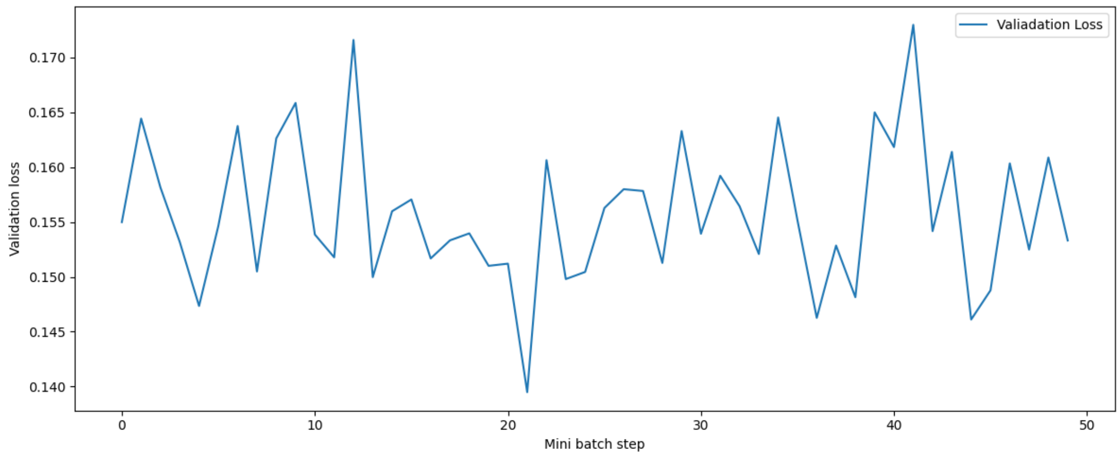
[9]:
```
correct = 0
total = 0
count = 0
validation_error = []
```

```python
prediction_accuracy = []
validation_running_loss = 0.0
model.eval()
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for i, data in enumerate(testloader):
        count+=1
        images, labels = data # Batch sized images and labels will be loaded here i.e, 4 images and 4 labels are loaded
        images = images.to(device)
        labels = labels.to(device)

        # forward pass
        outputs = model(images)

        # loss computation
        loss = criterion(outputs, labels)

        # Accumulate the loss value
        validation_running_loss += loss.item()

        predicted_probability, predicted = torch.max(outputs.data, 1) # each input image will have 10 probablities for classification,
                                                                      # from that we need to pick the maximum probability class as our predicted class

        total += labels.size(0)
        correct += (predicted == labels).sum().item()

        if ((i+1) % 50) == 0:    # print every 500 mini-batches
          avg_validating_loss = validation_running_loss / 500;
          print(f'[Step : {i + 1:5d}] , Average Validation loss: {avg_validating_loss}')
          validation_error.append(avg_validating_loss)
          validation_running_loss = 0.0

# Plot
plt.figure(figsize=(12, 5))
plt.plot(validation_error, label='Valiadation Loss')
plt.xlabel('Mini batch step')
plt.ylabel('Validation loss')
plt.legend()

plt.tight_layout()
plt.show()
```

```
print(f'\n Accuracy of the network on the {count*batch_size} test images: {100␣
↪* correct // total} %')
```

```
[Step :    50] , Average Validation loss: 0.1549882230758667
[Step :   100] , Average Validation loss: 0.16442338073253632
[Step :   150] , Average Validation loss: 0.15813990497589112
[Step :   200] , Average Validation loss: 0.15318271684646606
[Step :   250] , Average Validation loss: 0.14734534513950348
[Step :   300] , Average Validation loss: 0.15464624726772308
[Step :   350] , Average Validation loss: 0.16374450182914735
[Step :   400] , Average Validation loss: 0.15049061644077302
[Step :   450] , Average Validation loss: 0.16261842656135558
[Step :   500] , Average Validation loss: 0.16584826993942262
[Step :   550] , Average Validation loss: 0.1538564373254776
[Step :   600] , Average Validation loss: 0.15177999198436737
[Step :   650] , Average Validation loss: 0.17158732211589814
[Step :   700] , Average Validation loss: 0.1499726175069809
[Step :   750] , Average Validation loss: 0.1559773054122925
[Step :   800] , Average Validation loss: 0.1570502998828888
[Step :   850] , Average Validation loss: 0.15167849338054656
[Step :   900] , Average Validation loss: 0.15332641804218292
[Step :   950] , Average Validation loss: 0.15396223986148835
[Step :  1000] , Average Validation loss: 0.15100398313999175
[Step :  1050] , Average Validation loss: 0.15120305609703064
[Step :  1100] , Average Validation loss: 0.13949024152755737
[Step :  1150] , Average Validation loss: 0.16063215374946593
[Step :  1200] , Average Validation loss: 0.1497985657453537
[Step :  1250] , Average Validation loss: 0.1504457222223282
[Step :  1300] , Average Validation loss: 0.15627634048461914
[Step :  1350] , Average Validation loss: 0.1579882913827896
[Step :  1400] , Average Validation loss: 0.1578264102935791
[Step :  1450] , Average Validation loss: 0.15126684832572937
[Step :  1500] , Average Validation loss: 0.16328440058231353
[Step :  1550] , Average Validation loss: 0.15392676985263826
[Step :  1600] , Average Validation loss: 0.15921029889583588
[Step :  1650] , Average Validation loss: 0.15643921947479247
[Step :  1700] , Average Validation loss: 0.15208053719997405
[Step :  1750] , Average Validation loss: 0.1645223777294159
[Step :  1800] , Average Validation loss: 0.15516118913888932
[Step :  1850] , Average Validation loss: 0.14626326990127564
[Step :  1900] , Average Validation loss: 0.15285128700733186
[Step :  1950] , Average Validation loss: 0.14814420807361603
[Step :  2000] , Average Validation loss: 0.16498741912841797
[Step :  2050] , Average Validation loss: 0.16182583558559419
[Step :  2100] , Average Validation loss: 0.1729657131433487
[Step :  2150] , Average Validation loss: 0.15416736328601838
[Step :  2200] , Average Validation loss: 0.16138960337638855
```

```
[Step :  2250] , Average Validation loss: 0.14611303555965424
[Step :  2300] , Average Validation loss: 0.14876924884319306
[Step :  2350] , Average Validation loss: 0.16034159791469574
[Step :  2400] , Average Validation loss: 0.15248453748226165
[Step :  2450] , Average Validation loss: 0.16087620198726654
[Step :  2500] , Average Validation loss: 0.1533157056570053
```




```
Accuracy of the network on the 10000 test images: 46 %
```

Saving the Model

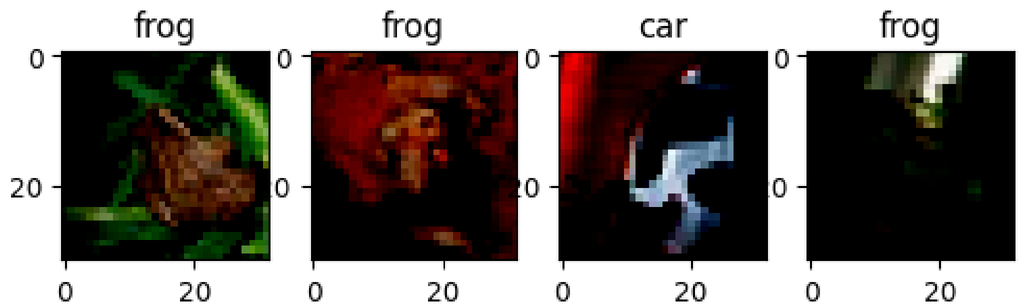
[10]:
```
PATH = './cifar_net.pth'
torch.save(model.state_dict(), PATH)
```

3. Checking the Ground Truth Values for the Test data

[11]:
```
test_data_itr = iter(testloader)
test_images, labels = next(test_data_itr)
test_images, labels = next(test_data_itr)
# print the test images wiht it's ground truth value
for i in range(test_images.size(0)):
  image = test_images.T[:,:,:,i]
  plt.subplot(1, batch_size, i+1)
  plt.imshow(image)
  plt.title(classes[labels[i]])
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```



Load the Trained Model

[12]:
```
saved_model = MLP_CIFR10(batch_norm=True).to(device)
saved_model.load_state_dict(torch.load(PATH))
```

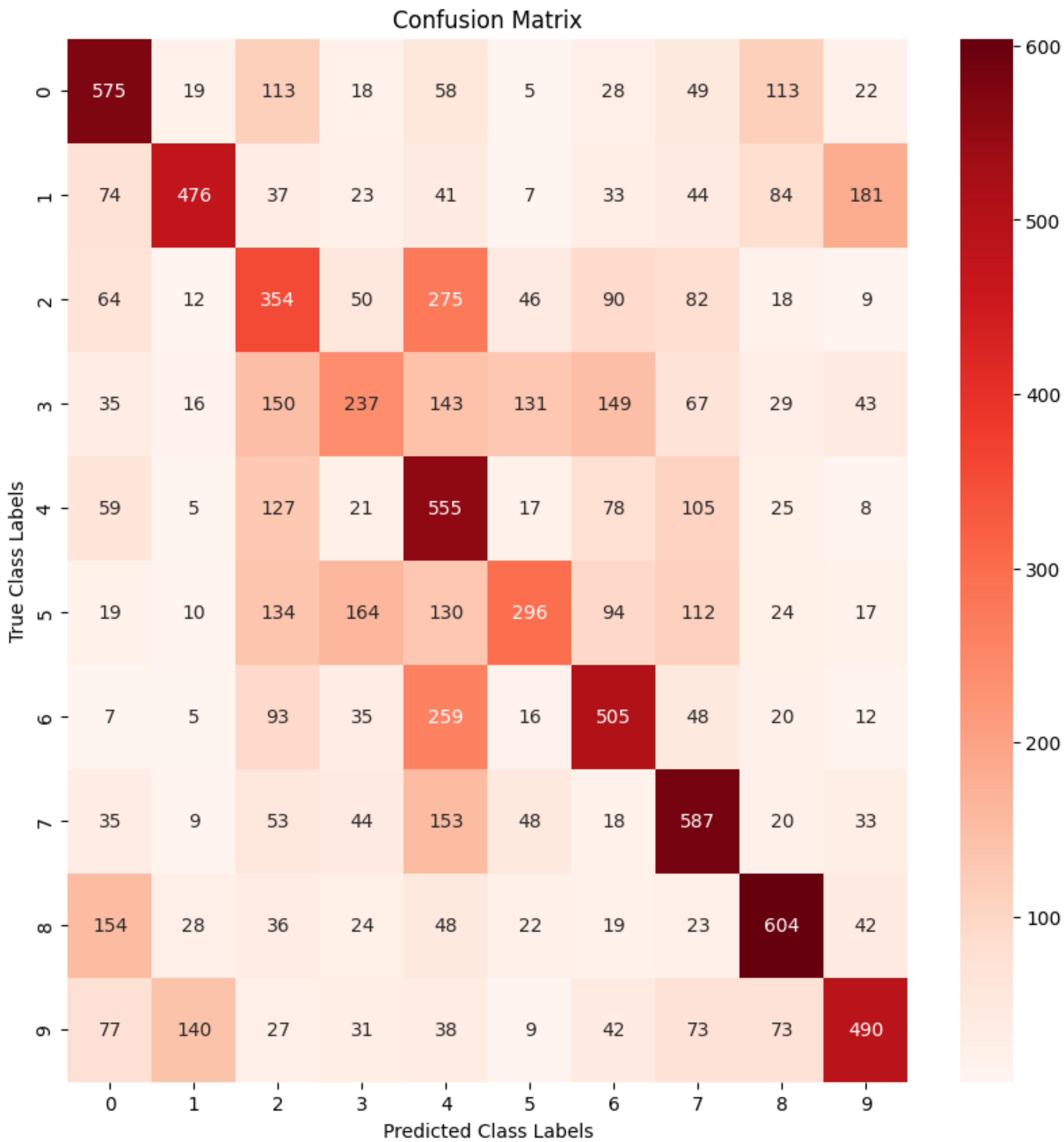
[12]:
```
<All keys matched successfully>
```

[13]:
```
trained_outputs = saved_model(images)
predicted_probability, predicted = torch.max(trained_outputs, 1)

print("Predicted classes : ",  [classes[predicted[idx]] for idx in␣
 ↪range(batch_size)])
```

```
Predicted classes :  ['frog', 'bird', 'car', 'horse']
```

[14]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns

true_labels_list = []
predicted_labels_list   = []
model.eval()

with torch.no_grad( ):
  for images, labels in testloader:
    images = images.to(device)
    labels = labels.to(device)
    output = model(images)
    predictions = F.softmax(output, dim=1)
    predicted_values, predicted_labels = torch.max(predictions, 1)
    true_labels_list.extend(labels.cpu().numpy())
    predicted_labels_list.extend(predicted_labels.cpu().numpy())

disp_matrix = confusion_matrix(true_labels_list, predicted_labels_list)
```

```python
# Plot the confusion matrix
plt.figure(figsize=(10, 10))
sns.heatmap(disp_matrix, annot=True, fmt="d", cmap="Reds", xticklabels=range(10), yticklabels=range(10))
plt.title("Confusion Matrix")
plt.xlabel("Predicted Class Labels")
plt.ylabel("True Class Labels")
plt.show()
```



Batch Normalization Decreased the Training Accuracy and Validation Accuracy, In general Applying Batch Normalization must result in faster convergence.