v3tk8ohpd

March 12, 2024

[243]:
```python
import os
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
from typing import OrderedDict
```

Defining class

[244]:
```python
class VGG11_CIFR10(nn.Module):
  def __init__(self, num_channels = 3, num_classes = 10):
    super(VGG11_CIFR10, self).__init__();
    self.num_channels = num_channels;
    self.num_classes = num_classes;
    # Convolution and Pooling
    self.convolution_layers = nn.Sequential(OrderedDict([
                                                        ('conv1', nn.
 ↪Conv2d(in_channels=num_channels, out_channels=64, kernel_size=3, padding=1)),
                                                        ('relu1', nn.
 ↪ReLU(inplace=True)),
                                                        ('Maxpool1', nn.
 ↪MaxPool2d(kernel_size=2,stride=2)), # Reducing the resulting image to half

                                                        ('conv2', nn.
 ↪Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)),
                                                        ('relu2', nn.
 ↪ReLU(inplace=True)),
                                                        ('Maxpool2', nn.
 ↪MaxPool2d(kernel_size=2,stride=2)), # Reducing the resulting image to half
```

```
                                                    ('conv3a', nn.
↪Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)),
                                                    ('relu3a', nn.
↪ReLU(inplace=True)),
                                                    ('conv3b', nn.
↪Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1)),
                                                    ('relu3b', nn.
↪ReLU(inplace=True)),
                                                    ('Maxpool3', nn.
↪MaxPool2d(kernel_size=2,stride=2)), # Reducing the resulting image to half

                                                    ('conv4a', nn.
↪Conv2d(in_channels=256, out_channels=512, kernel_size=3, padding=1)),
                                                    ('relu4a', nn.
↪ReLU(inplace=True)),
                                                    ('conv4b', nn.
↪Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)),
                                                    ('relu4b', nn.
↪ReLU(inplace=True)),
                                                    ('Maxpool4', nn.
↪MaxPool2d(kernel_size=2,stride=2)), # Reducing the resulting image to half

                                                    ('conv5a', nn.
↪Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)),
                                                    ('relu5a', nn.
↪ReLU(inplace=True)),
                                                    ('conv5b', nn.
↪Conv2d(in_channels=512, out_channels=512, kernel_size=3, padding=1)),
                                                    ('relu5b', nn.
↪ReLU(inplace=True)),
                                                    # Here the dimension of␣
↪the features will be 512 * 2 * 2
                                                    ('Maxpool5', nn.
↪MaxPool2d(kernel_size=2,stride=2)), # Reducing the resulting image to half
                                                    # after Maxpool5 the␣
↪dimension of the image becomes 512 * 1 * 1

                              ]))
  # Fully connected Layers
  self.fc = nn.Sequential(OrderedDict([
                                    ('fc1', nn.Linear(in_features=512*1*1,␣
↪out_features=4096)),
                                    ('relu1', nn.ReLU(inplace=True)),
                                    ('Dropout1', nn.Dropout(0.5)),
```

```
                                        ('fc2', nn.Linear(in_features=4096,␣
↪out_features=4096)),
                                        ('relu2', nn.ReLU(inplace=True)),
                                        ('Dropout2', nn.Dropout(0.5)),

                                        ('fc3', nn.Linear(in_features=4096,␣
↪out_features=1000)),
                                        ('relu3', nn.ReLU(inplace=True)),
                                        ('Dropout3', nn.Dropout(0.5)),

                                        ('fc_out', nn.Linear(in_features=1000,␣
↪out_features=10))
                                 ]))


# forward Prop
  def forward(self, x):
    x = self.convolution_layers(x);
    # flatten the data
    x = torch.flatten(x, 1);
    # Pass thorugh Fully connected layers
    x = self.fc(x);
    return x
```

[245]:

```
# Query for GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
print(torch.cuda.device_count())
print(torch.cuda.get_device_name(0))
```

```
cuda
1
Tesla P100-PCIE-16GB
```

[246]:

```
epochs = 80 # ideal = 59
batch_size = 128
learning_rate = 0.001
momentum = 0.9 # changed from 0.9 to 0.09
wt_decay = 0.0005
```

[247]:

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]) # Normalize the␣
↪input data to mean = 0.5 and SD = 0.5

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)

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)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[248]:
```
cnn_model = VGG11_CIFR10().to(device=device)

# Using Xavier initialization weights and Biases for convolutional layers
def weights_init(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
        init.xavier_uniform_(m.weight.data)
        if m.bias is not None:
            m.bias.data.zero_()

cnn_model.apply(weights_init)  # Enable this to initialize the weights

# for param in cnn_model.parameters():
#   print("Total number of parameters : ",param.numel())
#   print("Type : ", type(param), "\tSize : ", param.size())
```

[248]:
```
VGG11_CIFR10(
  (convolution_layers): Sequential(
    (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu1): ReLU(inplace=True)
    (Maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (relu2): ReLU(inplace=True)
    (Maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (conv3a): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (relu3a): ReLU(inplace=True)
    (conv3b): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (relu3b): ReLU(inplace=True)
    (Maxpool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
```

```
    (conv4a): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (relu4a): ReLU(inplace=True)
    (conv4b): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (relu4b): ReLU(inplace=True)
    (Maxpool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (conv5a): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (relu5a): ReLU(inplace=True)
    (conv5b): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (relu5b): ReLU(inplace=True)
    (Maxpool5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (fc): Sequential(
    (fc1): Linear(in_features=512, out_features=4096, bias=True)
    (relu1): ReLU(inplace=True)
    (Dropout1): Dropout(p=0.5, inplace=False)
    (fc2): Linear(in_features=4096, out_features=4096, bias=True)
    (relu2): ReLU(inplace=True)
    (Dropout2): Dropout(p=0.5, inplace=False)
    (fc3): Linear(in_features=4096, out_features=1000, bias=True)
    (relu3): ReLU(inplace=True)
    (Dropout3): Dropout(p=0.5, inplace=False)
    (fc_out): Linear(in_features=1000, out_features=10, bias=True)
  )
)
```

[249]:

```
#  loss function
criterion = nn.CrossEntropyLoss()

# the optimizer
optimizer = optim.SGD(cnn_model.parameters(), lr=learning_rate,␣
 ↪momentum=momentum, weight_decay=wt_decay, nesterov=True ) #  ,␣
 ↪weight_decay=wt_decay
                              # These parameters are taken from the section 3.1␣
 ↪of the paper https://arxiv.org/pdf/1409.1556.pdf on VGGNet
```

[250]:

```
training_error = []    # To record Training error through the training process
interm_accuracy = []  # To record the intermediate accuracy values for plotting
count = 0
total = 0
correct = 0
```

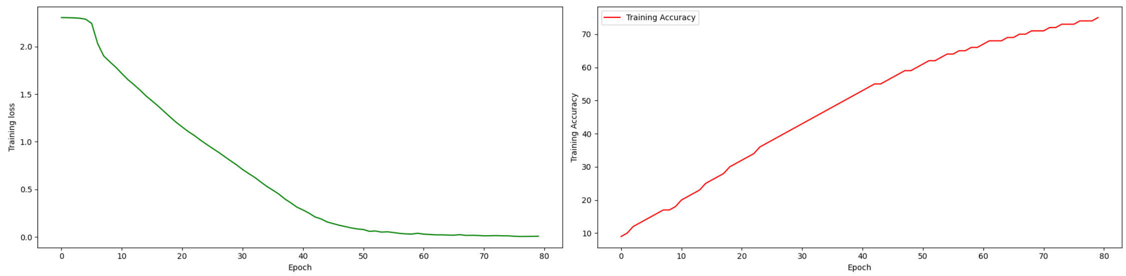
```python
for epoch in range(epochs):
  #set the model in training mode
  cnn_model.train()
  running_loss = 0.0
  interm_total = 0.0
  interm_correct = 0.0

  for i, data in enumerate(trainloader):
    count = count + 1
    inputs, labels = data
    inputs = inputs.to(device)  # To perform on the available GPU
    labels = labels.to(device)

    # reset the grads to Zero
    optimizer.zero_grad()

    # forward pass
    outputs = cnn_model(inputs)

    # calculate loss
    loss = criterion(outputs, labels)

    # Back prop
    loss.backward()

    # Gradient update
    optimizer.step()

    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()

    interm_total += labels.size(0)
    interm_correct += (predicted == labels).sum().item()

  avg_training_loss = running_loss / len(trainloader);
  print(f'[Epoch : {epoch + 1} , Average Training loss: {avg_training_loss:.
↪6f}') # running_loss / 2000:.3f
  training_error.append(avg_training_loss)
  interm_accuracy.append((100 * correct)//total)
  running_loss = 0.0
```

```python
# Plot the computed stats
fig=plt.figure(figsize=(20, 5))

fig.add_subplot(1,2,1)
plt.plot(training_error, label='Training Loss', color='g')
plt.xlabel('Epoch')
plt.ylabel('Training loss')

fig.add_subplot(1,2,2)
plt.plot(interm_accuracy, label='Training Accuracy', color='r')
plt.xlabel('Epoch')
plt.ylabel('Training Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

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

```
[Epoch : 1 , Average Training loss: 2.302522
[Epoch : 2 , Average Training loss: 2.301532
[Epoch : 3 , Average Training loss: 2.299735
[Epoch : 4 , Average Training loss: 2.295923
[Epoch : 5 , Average Training loss: 2.285982
[Epoch : 6 , Average Training loss: 2.241518
[Epoch : 7 , Average Training loss: 2.029145
[Epoch : 8 , Average Training loss: 1.899869
[Epoch : 9 , Average Training loss: 1.839308
[Epoch : 10 , Average Training loss: 1.781896
[Epoch : 11 , Average Training loss: 1.715705
[Epoch : 12 , Average Training loss: 1.652579
[Epoch : 13 , Average Training loss: 1.599071
[Epoch : 14 , Average Training loss: 1.542697
[Epoch : 15 , Average Training loss: 1.481198
[Epoch : 16 , Average Training loss: 1.428835
[Epoch : 17 , Average Training loss: 1.375176
[Epoch : 18 , Average Training loss: 1.317435
[Epoch : 19 , Average Training loss: 1.259905
[Epoch : 20 , Average Training loss: 1.204048
[Epoch : 21 , Average Training loss: 1.155281
[Epoch : 22 , Average Training loss: 1.107426
[Epoch : 23 , Average Training loss: 1.066233
[Epoch : 24 , Average Training loss: 1.018204
[Epoch : 25 , Average Training loss: 0.974569
[Epoch : 26 , Average Training loss: 0.932381
[Epoch : 27 , Average Training loss: 0.890616
[Epoch : 28 , Average Training loss: 0.845757
```

```
[Epoch : 29 , Average Training loss: 0.801019
[Epoch : 30 , Average Training loss: 0.758141
[Epoch : 31 , Average Training loss: 0.709282
[Epoch : 32 , Average Training loss: 0.667232
[Epoch : 33 , Average Training loss: 0.627070
[Epoch : 34 , Average Training loss: 0.579211
[Epoch : 35 , Average Training loss: 0.532675
[Epoch : 36 , Average Training loss: 0.493570
[Epoch : 37 , Average Training loss: 0.451655
[Epoch : 38 , Average Training loss: 0.400451
[Epoch : 39 , Average Training loss: 0.359317
[Epoch : 40 , Average Training loss: 0.314668
[Epoch : 41 , Average Training loss: 0.284905
[Epoch : 42 , Average Training loss: 0.251763
[Epoch : 43 , Average Training loss: 0.211887
[Epoch : 44 , Average Training loss: 0.191113
[Epoch : 45 , Average Training loss: 0.159688
[Epoch : 46 , Average Training loss: 0.141781
[Epoch : 47 , Average Training loss: 0.124494
[Epoch : 48 , Average Training loss: 0.111230
[Epoch : 49 , Average Training loss: 0.096584
[Epoch : 50 , Average Training loss: 0.085896
[Epoch : 51 , Average Training loss: 0.080077
[Epoch : 52 , Average Training loss: 0.061010
[Epoch : 53 , Average Training loss: 0.064592
[Epoch : 54 , Average Training loss: 0.052739
[Epoch : 55 , Average Training loss: 0.055744
[Epoch : 56 , Average Training loss: 0.048426
[Epoch : 57 , Average Training loss: 0.039234
[Epoch : 58 , Average Training loss: 0.034343
[Epoch : 59 , Average Training loss: 0.032113
[Epoch : 60 , Average Training loss: 0.040394
[Epoch : 61 , Average Training loss: 0.031345
[Epoch : 62 , Average Training loss: 0.028019
[Epoch : 63 , Average Training loss: 0.023974
[Epoch : 64 , Average Training loss: 0.023921
[Epoch : 65 , Average Training loss: 0.021788
[Epoch : 66 , Average Training loss: 0.020803
[Epoch : 67 , Average Training loss: 0.025842
[Epoch : 68 , Average Training loss: 0.018407
[Epoch : 69 , Average Training loss: 0.019480
[Epoch : 70 , Average Training loss: 0.017701
[Epoch : 71 , Average Training loss: 0.013626
[Epoch : 72 , Average Training loss: 0.014620
[Epoch : 73 , Average Training loss: 0.016189
[Epoch : 74 , Average Training loss: 0.014178
[Epoch : 75 , Average Training loss: 0.013496
[Epoch : 76 , Average Training loss: 0.009316
```

```
[Epoch : 77 , Average Training loss: 0.006786
[Epoch : 78 , Average Training loss: 0.007772
[Epoch : 79 , Average Training loss: 0.008155
[Epoch : 80 , Average Training loss: 0.009443
```



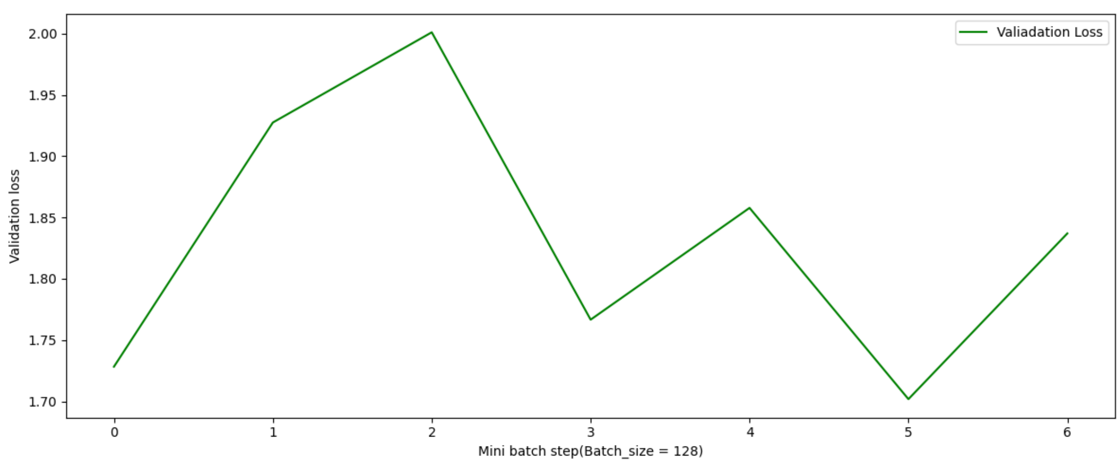

```
Training Finished with accuracy :  75
```

[251]:

```python
correct = 0
total = 0
count = 0
validation_error = []
prediction_accuracy = []
validation_running_loss = 0.0
cnn_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

        images = images.to(device)  # To perform on the available GPU if present else on cpu
        labels = labels.to(device)

        # calculate outputs by running images through the network
        outputs = cnn_model(images) # forward pass

        loss = criterion(outputs, labels)
        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
```

```python
        total += labels.size(0)
        # print((predicted == labels))
        correct += (predicted == labels).sum().item()
        # print(total , " , ",correct)
        if ((i+1) % 10) == 0:
          # print("inside loop")
          avg_validating_loss = validation_running_loss / 10;
          print(f'[Step : {i + 1:5d}] , Average Validation loss:␣
↪{avg_validating_loss}') # running_loss / 2000:.3f
          validation_error.append(avg_validating_loss)
          validation_running_loss = 0.0

# Plot the stats recoreded in the above steps
plt.figure(figsize=(12, 5))
plt.plot(validation_error, label='Valiadation Loss', color = 'g')
plt.xlabel(f'Mini batch step(Batch_size = {batch_size})')
plt.ylabel('Validation loss')
plt.legend()

plt.tight_layout()
plt.show()

print(f'\n Accuracy of the network on the 10000 test images: {100 * correct //␣
↪total} %')
```

```
[Step :    10] , Average Validation loss: 1.7283137679100036
[Step :    20] , Average Validation loss: 1.9274780392646789
[Step :    30] , Average Validation loss: 2.0010708570480347
[Step :    40] , Average Validation loss: 1.766665530204773
[Step :    50] , Average Validation loss: 1.8578355431556701
[Step :    60] , Average Validation loss: 1.7018407464027405
[Step :    70] , Average Validation loss: 1.836998736858368
```



```
Accuracy of the network on the 10000 test images: 73 %
```

[252]:
```
#save the model for inference
PATH = './cifar10_vgg_net.pth'
torch.save(cnn_model.state_dict(), PATH)
```

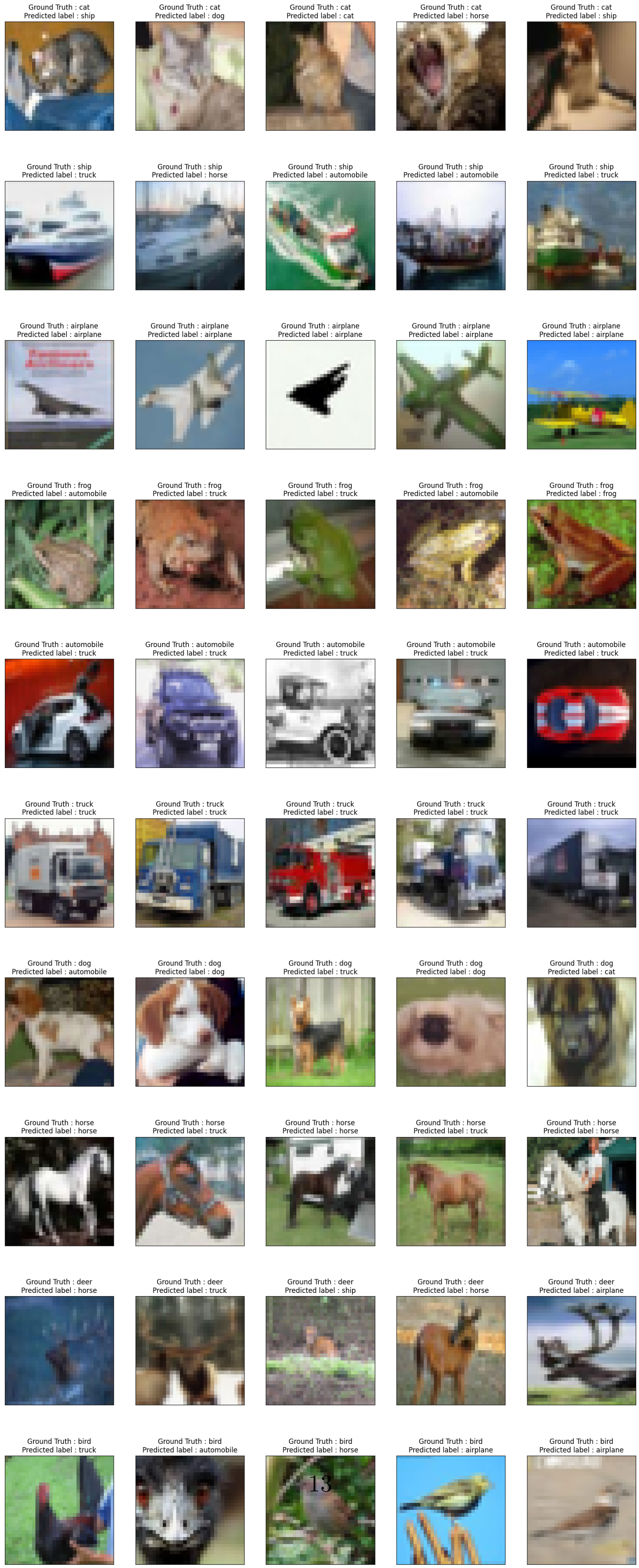
[253]:
```
# Compare the results of the Ground Truth vs Prediction
saved_model =  VGG11_CIFR10().to(device)
# Load the CNN Model
saved_model.load_state_dict(torch.load(PATH))
```

[253]:
```
<All keys matched successfully>
```

[254]:
```
from itertools import count
#since data shuffling has been turned on earlier, now we will load the data␣
 ↪using pickle to verify the images class wise
def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='latin1')
    return dict

file = r'./data/cifar-10-batches-py/test_batch'
meta_file = r'./data/cifar-10-batches-py/batches.meta'
meta_data = unpickle(meta_file)
ground_truth = unpickle(file)

# take the images data from batch data
images = ground_truth['data']
# reshape and transpose the images
# print("images type : ", type(images), "shape : ", images.shape)
images = images.reshape(len(images),3,32,32).transpose(0,2,3,1)
# take labels of the images
labels = ground_truth['labels']
# label names of the images
label_names = meta_data['label_names']

print(type(label_names),label_names)

rows, columns = 10, 5
# print(type(images), images.shape)
tmp_dict = dict()
# pick 5 images from each class from test_batch
for i, label in enumerate(labels):
```

```
    if len(tmp_dict.setdefault(label, [])) < 5:
        tmp_dict[label].append(i)

imageId =  np.array(list(tmp_dict.values())).flatten()

# print('Generated Image ID : ', imageId, '\nShape : ', imageId.shape)

# take images for above random image ids
images = images[imageId]
# take labels for these images only
labels = [labels[i] for i in imageId]

images_torch = torch.from_numpy(images.astype(np.float32)).to(device)
images_torch = images_torch.permute(0, 3, 1, 2)

# print("images: ", type(images_torch), " , ", images_torch.shape)
trained_outputs = saved_model(images_torch)
print(type(trained_outputs), trained_outputs.shape)
predicted_probability, predicted = torch.max(trained_outputs, 1)

# define figure
fig=plt.figure(figsize=(20, 50))
# visualize these random images
for i, pred_val in enumerate(predicted):
    fig.add_subplot(rows, columns, i+1)
    plt.imshow(images[i])
    plt.xticks([])
    plt.yticks([])
    plt.title("{}"
          .format(f'Ground Truth : {label_names[labels[i]]}\nPredicted label :␣
 ↪{label_names[pred_val.item()]}'))

plt.show()
plt.subplots_adjust(hspace=0.5)
```

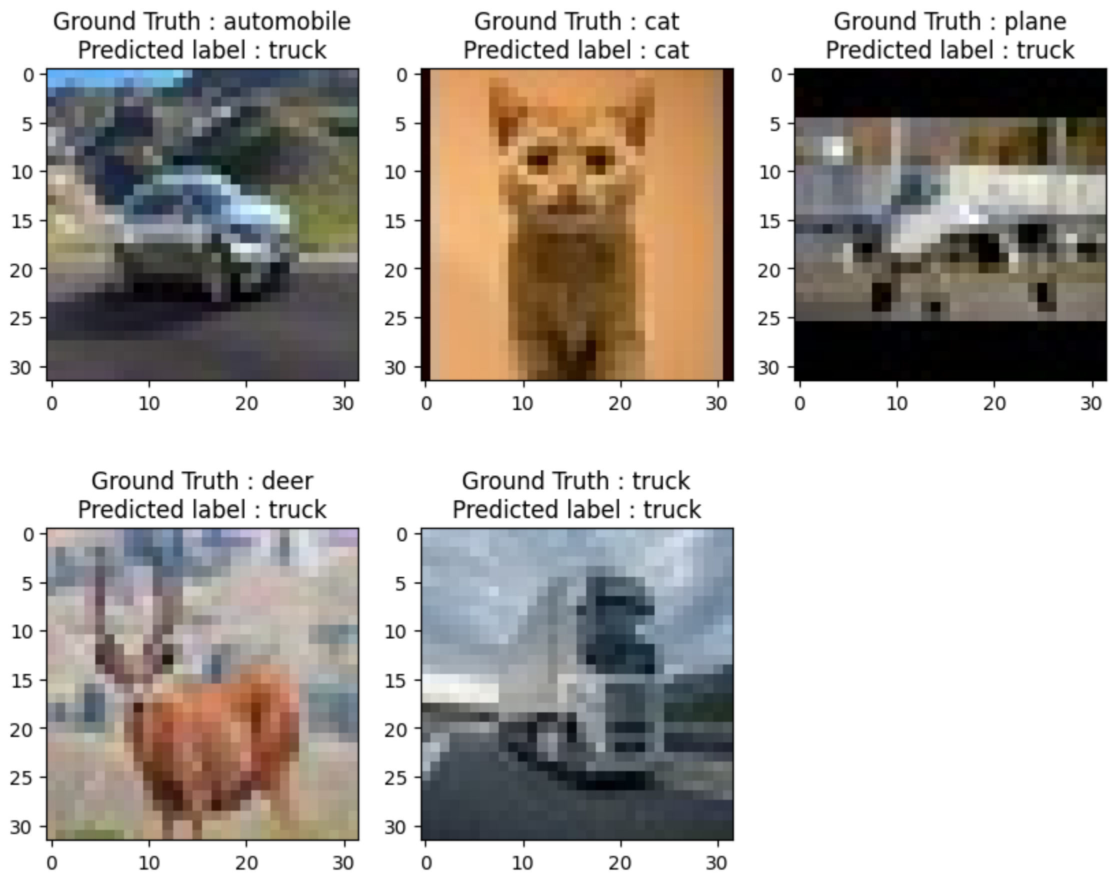
```
<class 'list'> ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck']
<class 'torch.Tensor'> torch.Size([50, 10])
```


Ground Truth : cat
Predicted label : ship
Ground Truth : cat
Predicted label : dog
Ground Truth : cat
Predicted label : cat
Ground Truth : cat
Predicted label : horse
Ground Truth : cat
Predicted label : ship
Ground Truth : ship
Predicted label : truck
Ground Truth : ship
Predicted label : horse
Ground Truth : ship
Predicted label : automobile
Ground Truth : ship
Predicted label : automobile
Ground Truth : ship
Predicted label : truck
Ground Truth : airplane
Predicted label : airplane
Ground Truth : airplane
Predicted label : airplane
Ground Truth : airplane
Predicted label : airplane
Ground Truth : airplane
Predicted label : airplane
Ground Truth : airplane
Predicted label : airplane
Ground Truth : frog
Predicted label : automobile
Ground Truth : frog
Predicted label : truck
Ground Truth : frog
Predicted label : truck
Ground Truth : frog
Predicted label : automobile
Ground Truth : frog
Predicted label : frog
Ground Truth : automobile
Predicted label : truck
Ground Truth : automobile
Predicted label : truck
Ground Truth : automobile
Predicted label : truck
Ground Truth : automobile
Predicted label : truck
Ground Truth : automobile
Predicted label : truck
Ground Truth : truck
Predicted label : truck
Ground Truth : truck
Predicted label : truck
Ground Truth : truck
Predicted label : truck
Ground Truth : truck
Predicted label : truck
Ground Truth : truck
Predicted label : truck
Ground Truth : dog
Predicted label : automobile
Ground Truth : dog
Predicted label : dog
Ground Truth : dog
Predicted label : truck
Ground Truth : dog
Predicted label : dog
Ground Truth : dog
Predicted label : cat
Ground Truth : horse
Predicted label : horse
Ground Truth : horse
Predicted label : truck
Ground Truth : horse
Predicted label : horse
Ground Truth : horse
Predicted label : truck
Ground Truth : horse
Predicted label : horse
Ground Truth : deer
Predicted label : horse
Ground Truth : deer
Predicted label : truck
Ground Truth : deer
Predicted label : ship
Ground Truth : deer
Predicted label : horse
Ground Truth : deer
Predicted label : airplane
Ground Truth : bird
Predicted label : truck
Ground Truth : bird
Predicted label : automobile
Ground Truth : bird
Predicted label : horse
Ground Truth : bird
Predicted label : airplane
Ground Truth : bird
Predicted label : airplane

<Figure size 640x480 with 0 Axes>

[255]:
```python
random_img_path = '/kaggle/input/cnns-dataset/'
labels = ['automobile', 'cat', 'plane', 'deer', 'truck']
predicted_labels = []
images_list = os.listdir(random_img_path)
all_image_data = []

reloaded_model =  VGG11_CIFR10().to(device)
reloaded_model.load_state_dict(torch.load('./cifar10_vgg_net.pth'))
reloaded_model.eval()

fig=plt.figure(figsize=(10, 8))

# add images to a np-array
for image in images_list:
    img = plt.imread(random_img_path+image)
    all_image_data.append(img)

concat_images = np.stack(all_image_data, axis = 0)
concat_images = concat_images.transpose(0, 3, 1,2)

images_torch = torch.from_numpy(concat_images.astype(np.float32)).to(device)
trained_outputs = saved_model(images_torch)
predicted_probability, predicted = torch.max(trained_outputs, 1)
# print(type(predicted),predicted.shape)


for i, image in enumerate(images_list):
#     print(image)
    fig.add_subplot(2,3,i+1)
    img = plt.imread(random_img_path+image)
    plt.imshow(img)

    #perform classification
    img_torch = torch.from_numpy(img.astype(np.float32)).to(device)
#     print(type(img_torch), img_torch.shape)
    img_torch = img_torch.permute(2,0,1)
#     print(type(img_torch), img_torch.shape)
    outs = reloaded_model(images_torch)
#     print(type(outs), outs.shape)
    new_predicted_probability, new_predicted = torch.max(outs, 1)
#     print(type(new_predicted), new_predicted.shape, new_predicted)
    plt.title("{}"
```

```
        .format(f'Ground Truth : {labels[i]}\nPredicted label : {label_names[predicted[i]]}'))

plt.show()
plt.subplots_adjust(hspace=0.5)
```




```
<Figure size 640x480 with 0 Axes>
```

[ ]: