CIFAR10 Image Classification Comparing MLP vs Linear Models

Syed Arham Naqvi (100590852)

March 8, 2025

[134]:
```
import torch
import torchvision
import torchvision.transforms as transforms
```

[135]:
```
# GPU usage
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Assuming that we are on a CUDA machine, this should print a CUDA device:
print(device)
```

```
cuda
```

[136]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
    #  transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))␣
 ↪# true mean and std for each channel
    # ])
```

[137]:
```
batch_size = 4

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)
```

```
Files already downloaded and verified
```

[138]:
```
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)
```

```
Files already downloaded and verified
```

[139]:
```
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck')
```

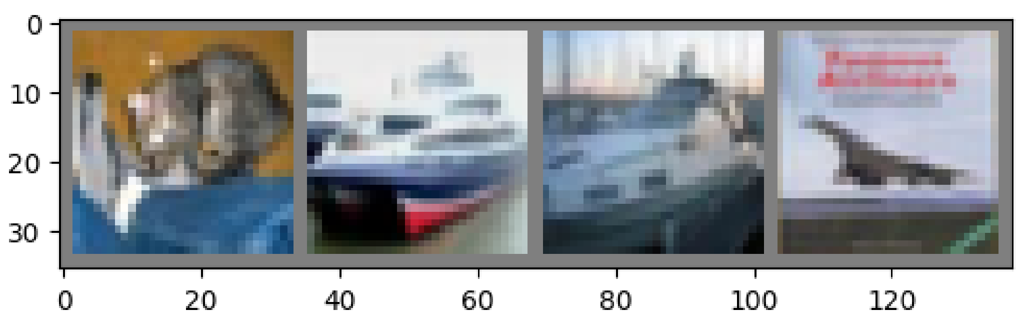
[140]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[141]:
```
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

1 Test Images

[172]:
```
dataiter = iter(testloader)
t_images, t_labels = next(dataiter)
t_images, t_labels = t_images.to(device), t_labels.to(device)

# print images
imshow(torchvision.utils.make_grid(t_images.to('cpu')))
print('GroundTruth: ', ' '.join(f'{classes[t_labels[j]]:5s}' for j in range(4)))
```




```
GroundTruth:  cat   ship  ship  plane
```

2 10-way linear model

[143]:
```
import torch.nn as nn
import torch.nn.functional as F

class ALinearModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(32*32*3, 10)

    def forward(self, x):
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = self.fc(x)
        return x

linear_model = ALinearModel().to(device)
```

3 Training (Linear Model)

[144]:
```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(linear_model.parameters(), lr=0.001, momentum=0.9)
```

[145]:
```
for epoch in range(4):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = linear_model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0
```

```python
print('Finished Training')
```

```
[1,  2000] loss: 2.188
[1,  4000] loss: 2.166
[1,  6000] loss: 2.145
[1,  8000] loss: 2.140
[1, 10000] loss: 2.171
[1, 12000] loss: 2.157
[2,  2000] loss: 2.086
[2,  4000] loss: 2.105
[2,  6000] loss: 2.124
[2,  8000] loss: 2.095
[2, 10000] loss: 2.104
[2, 12000] loss: 2.121
[3,  2000] loss: 2.022
[3,  4000] loss: 2.044
[3,  6000] loss: 2.099
[3,  8000] loss: 2.084
[3, 10000] loss: 2.131
[3, 12000] loss: 2.101
[4,  2000] loss: 2.045
[4,  4000] loss: 2.074
[4,  6000] loss: 2.068
[4,  8000] loss: 2.067
[4, 10000] loss: 2.058
[4, 12000] loss: 2.105
Finished Training
```

4 Saving (Linear Model)

[173]:
```python
PATH = './cifar_linear_model.pth'
torch.save(linear_model.state_dict(), PATH)
```

5 Inference on Test Images (Linear Model)

[174]:
```python
linear_model = ALinearModel().to(device)
linear_model.load_state_dict(torch.load(PATH))
```

```
/tmp/ipykernel_14628/4251658964.py:2: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
```

```
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  linear_model.load_state_dict(torch.load(PATH))
```

[174]:
```
<All keys matched successfully>
```

[178]:
```
outputs = linear_model(t_images)
```

[180]:
```
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                              for j in range(4)))
```

```
Predicted:  dog   car   car   car
```

6 Test Set Accuracy (Linear Model)

[181]:
```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our␣
 ↪outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        # calculate outputs by running images through the network
        outputs = linear_model(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

```
Accuracy of the network on the 10000 test images: 11 %
```

7 Test Set Class-Wise Accuracy (Linear Model)

[182]:
```
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}
```

```
# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = linear_model(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

```
Accuracy for class: plane is 13.8 %
Accuracy for class: car   is 10.6 %
Accuracy for class: bird  is 33.9 %
Accuracy for class: cat   is 11.3 %
Accuracy for class: deer  is 15.6 %
Accuracy for class: dog   is 7.8 %
Accuracy for class: frog  is 2.7 %
Accuracy for class: horse is 6.2 %
Accuracy for class: ship  is 4.6 %
Accuracy for class: truck is 5.3 %
```

8 MLP Model

[152]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MyMLP(nn.Module):
    def __init__(self):
        super(MyMLP, self).__init__()

        # ----------------------------------------------------------
        # Input: 32×32×3 = 3072 elements per CIFAR-10 image.
        # Architecture:
        #   fc1: 3072 -> 2048
        #   fc2: 2048 -> 1024
        #   fc3: 1024 -> 512
        #   fc4: 512  -> 256
```

```
        #   fc5: 256  -> 10
        #
        # Applying a 20% dropout in four hidden layers when if enabled.
        # ----------------------------------------------------------

        self.fc1 = nn.Linear(3 * 32 * 32, 2048)
        self.drop1 = nn.Dropout(0.2)  # 20% dropout
        self.fc2 = nn.Linear(2048, 1024)
        self.drop2 = nn.Dropout(0.2)
        self.fc3 = nn.Linear(1024, 512)
        self.drop3 = nn.Dropout(0.2)
        self.fc4 = nn.Linear(512, 256)
        self.drop4 = nn.Dropout(0.2)
        self.fc5 = nn.Linear(256, 10)

    def forward(self, x, dropout=True):
        # Flatten image from (N, 3, 32, 32) -> (N, 3072)
        x = torch.flatten(x, start_dim=1)

        # fc1 -> ReLU
        x = F.relu(self.fc1(x))
        if dropout:
            x = self.drop1(x)

        # fc2 -> ReLU
        x = F.relu(self.fc2(x))
        if dropout:
            x = self.drop2(x)

        # fc3 -> ReLU
        x = F.relu(self.fc3(x))
        if dropout:
            x = self.drop3(x)

        # fc4 -> ReLU
        x = F.relu(self.fc4(x))
        if dropout:
            x = self.drop4(x)

        # Output layer (raw logits)
        x = self.fc5(x)
        return x

mlp_model = MyMLP().to(device)
```

9 Training (MLP Model)

[153]:
```
optimizer = optim.SGD(mlp_model.parameters(), lr=0.001, momentum=0.9)

for epoch in range(4):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = mlp_model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 2.170
[1,  4000] loss: 1.895
[1,  6000] loss: 1.772
[1,  8000] loss: 1.703
[1, 10000] loss: 1.645
[1, 12000] loss: 1.626
[2,  2000] loss: 1.549
[2,  4000] loss: 1.548
[2,  6000] loss: 1.509
[2,  8000] loss: 1.515
[2, 10000] loss: 1.505
[2, 12000] loss: 1.490
[3,  2000] loss: 1.416
[3,  4000] loss: 1.418
[3,  6000] loss: 1.386
[3,  8000] loss: 1.411
[3, 10000] loss: 1.400
[3, 12000] loss: 1.396
[4,  2000] loss: 1.310
```

```
[4,  4000] loss: 1.335
[4,  6000] loss: 1.310
[4,  8000] loss: 1.336
[4, 10000] loss: 1.345
[4, 12000] loss: 1.308
Finished Training
```

10 Saving (MLP Model)

[183]:
```python
PATH = './cifar_mlp_model.pth'
torch.save(mlp_model.state_dict(), PATH)
```

11 Inference on Test Images (MLP MODEL)

[184]:
```python
mlp_model = MyMLP().to(device)
mlp_model.load_state_dict(torch.load(PATH))
```

```
/tmp/ipykernel_14628/325627532.py:2: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  mlp_model.load_state_dict(torch.load(PATH))
```

[184]:
```
<All keys matched successfully>
```

[185]:
```python
outputs = mlp_model(t_images, False) # no need for dropof during inference
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                              for j in range(4)))
```

```
Predicted:  cat   ship  plane plane
```

12 Test Set Accuracy (MLP Model)

[186]:
```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our␣
↪outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        # calculate outputs by running images through the network
        outputs = mlp_model(images, False)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

```
Accuracy of the network on the 10000 test images: 51 %
```

13 Test Set Class-Wise Accuracy (MLP Model)

[187]:
```
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = mlp_model(images, False)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

```
Accuracy for class: plane is 58.5 %
Accuracy for class: car   is 60.9 %
Accuracy for class: bird  is 36.7 %
Accuracy for class: cat   is 21.7 %
Accuracy for class: deer  is 43.1 %
Accuracy for class: dog   is 36.3 %
Accuracy for class: frog  is 58.4 %
Accuracy for class: horse is 72.5 %
Accuracy for class: ship  is 72.4 %
Accuracy for class: truck is 52.4 %
```

We can observe clearly that utilizing an MLP model with dropout layers of $p = 0.2$ for improved generalizations results in much better accuracy than a simple linear model.