vgg-16

April 9, 2025

[1]:
```
import pandas as pd
import numpy as np
import torch
import os
torch.set_num_threads(os.cpu_count())
from torch import optim as optim
from torch import nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms
import random
np.random.seed(0)
torch.manual_seed(0)
random.seed(0)
os.environ['CUDA_LAUNCH_BLOCKING'] = '1'
```

Reference: https://www.kaggle.com/code/blurredmachine/vggnet-16-architecture-a-complete-guide

[2]:
```
data_folder = os.listdir("/kaggle/input/arsl-256")
num_classes = data_folder.__len__()
print(num_classes)
```

```
31
```

[12]:
```
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, Dataset
from PIL import Image

class ASLDataset(Dataset):
    def __init__(self, data_dir, transform=None):
        self.data_dir = data_dir
        self.transform = transform
        self.classes = os.listdir(data_dir)
        self.image_paths = []
        self.labels = []
        for i, class_name in enumerate(self.classes):
            class_dir = os.path.join(data_dir, class_name)
```

```
            for image_name in os.listdir(class_dir):
                image_path = os.path.join(class_dir, image_name)
                self.image_paths.append(image_path)
                self.labels.append(i)  # Use the class index as the label

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        image = Image.open(image_path).convert("RGB")
        label = self.labels[idx]

        if self.transform:
            image = self.transform(image)

        return image, label


transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

dataset = ASLDataset("/kaggle/input/arsl-256", transform=transform)
```

[13]:
```
from torch.utils.data import random_split

train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True,␣
 ↪num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False,␣
 ↪num_workers=2)
```

[14]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
Using device: cuda
```

1 Model

[15]:
```
class VGG(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU())
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU())
        self.layer4 = nn.Sequential(
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer5 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU())
        self.layer6 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU())
        self.layer7 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer8 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU())
        self.layer9 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU())
        self.layer10 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer11 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU())
        self.layer12 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU())
        self.layer13 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
```

```
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.fc = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(7*7*512, 4096),
            nn.ReLU())
        self.fc1 = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU())
        self.fc2= nn.Sequential(
            nn.Linear(4096, num_classes))

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        x = self.layer5(x)
        x = self.layer6(x)
        x = self.layer7(x)
        x = self.layer8(x)
        x = self.layer9(x)
        x = self.layer10(x)
        x = self.layer11(x)
        x = self.layer12(x)
        x = self.layer13(x)
        x = x.reshape(x.size(0), -1)
        x = self.fc(x)
        x = self.fc1(x)
        x = self.fc2(x)
        return x
```

[22]:
```
model = VGG(num_classes=num_classes).to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
from tqdm import tqdm
loss_fn = nn.CrossEntropyLoss()
```

[18]:
```
def train_model(model, optimizer, training_loader, criterion=loss_fn,␣
 ↪no_epochs=3):
    model.train()
    batches = []
    losses = []
    j = 0

    for epoch in range(no_epochs):  # Don't wrap this with tqdm
        running_loss = 0
```

```
        correct = 0
        total = 0

        loop = tqdm(enumerate(training_loader), total=len(training_loader),␣
↪desc=f"Epoch {epoch+1}/{no_epochs}")

        for i, (images, labels) in loop:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            # Update tqdm with current metrics
            loop.set_postfix(loss=loss.item(), accuracy=100 * correct / total)

            if i % 100 == 99:
                avg_loss = running_loss / 100
                losses.append(avg_loss)
                j += i
                batches.append(j)
                print(f"Epoch: {epoch}, Batch: {i+1}, Loss: {avg_loss:.3f},␣
↪Accuracy: {100 * correct / total:.2f}%")
                running_loss = 0

        if epoch % 2 == 0:
            print(f"Epoch {epoch+1} completed")

    return model, losses, batches
```

[19]:
```
def plot_loss(losses, batches):
    plt.plot(batches, losses)
    plt.xlabel('Batches')
    plt.ylabel('Loss')
    plt.title('Loss vs. Batches')
    plt.show()
```

[20]:
```
model1, losses, batches = train_model(model, optimizer, train_loader, loss_fn,␣
↪no_epochs=10)
```

```
Epoch 1/10:  25%|          | 100/393 [00:27<01:15,  3.87it/s, accuracy=4.44,
loss=3.41]
```

```
Epoch: 0, Batch: 100, Loss: 3.441, Accuracy: 4.44%

Epoch 1/10:  51%|        | 200/393 [00:53<00:50,  3.80it/s, accuracy=4.31,
loss=3.44]

Epoch: 0, Batch: 200, Loss: 3.432, Accuracy: 4.31%

Epoch 1/10:  76%|       | 300/393 [01:19<00:24,  3.78it/s, accuracy=4.02,
loss=3.44]

Epoch: 0, Batch: 300, Loss: 3.432, Accuracy: 4.02%

Epoch 1/10:  96%|      | 379/393 [01:41<00:03,  3.74it/s, accuracy=3.92,
loss=3.41]
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-20-848e53a6044e> in <cell line: 1>()
----> 1 model1, losses, batches = train_model(model, optimizer, train_loader,␣
 ↪loss_fn, no_epochs=10)

<ipython-input-18-8a2a2bc3b304> in train_model(model, optimizer,␣
 ↪training_loader, criterion, no_epochs)
     20             outputs = model(images)
     21             loss = criterion(outputs, labels)
---> 22             loss.backward()
     23             optimizer.step()
     24

/usr/local/lib/python3.10/dist-packages/torch/_tensor.py in backward(self,␣
 ↪gradient, retain_graph, create_graph, inputs)
    579                 inputs=inputs,
    580             )
--> 581         torch.autograd.backward(
    582             self, gradient, retain_graph, create_graph, inputs=inputs
    583         )

/usr/local/lib/python3.10/dist-packages/torch/autograd/__init__.py in␣
 ↪backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables,␣
 ↪inputs)
    345     # some Python versions print out the first line of a multi-line␣
 ↪function
    346     # calls in the traceback and some print out the last line
--> 347     _engine_run_backward(
    348         tensors,
    349         grad_tensors_,

/usr/local/lib/python3.10/dist-packages/torch/autograd/graph.py in␣
 ↪_engine_run_backward(t_outputs, *args, **kwargs)
```

```
    823         unregister_hooks = _register_logging_hooks_on_whole_graph(t_outputs)
    824     try:
--> 825         return Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward pass
    826             t_outputs, *args, **kwargs
    827         )  # Calls into the C++ engine to run the backward pass

KeyboardInterrupt:
```

accuracy doesn't change

2 Model 2 with BR

[ ]:
```
model2 = VGG16(num)
```

[23]:
```
class VGG16(nn.Module):
    def __init__(self, num_classes=10):
        super(VGG16, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU())
        self.layer2 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer3 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU())
        self.layer4 = nn.Sequential(
            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer5 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU())
        self.layer6 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU())
        self.layer7 = nn.Sequential(
```

```
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer8 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU())
        self.layer9 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU())
        self.layer10 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.layer11 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU())
        self.layer12 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU())
        self.layer13 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = 2, stride = 2))
        self.fc = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(7*7*512, 4096),
            nn.ReLU())
        self.fc1 = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(4096, 4096),
            nn.ReLU())
        self.fc2= nn.Sequential(
            nn.Linear(4096, num_classes))

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.layer5(out)
```

```
        out = self.layer6(out)
        out = self.layer7(out)
        out = self.layer8(out)
        out = self.layer9(out)
        out = self.layer10(out)
        out = self.layer11(out)
        out = self.layer12(out)
        out = self.layer13(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        out = self.fc1(out)
        out = self.fc2(out)
        return out
```

[ ]:
```
model2 = VGG16
```

[ ]:
```
def evaluate_model(model, loader):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in loader:
            images, labels = images.to(device), labels.to(device)

            outputs = model(images)

            _, predicted = torch.max(outputs.data, 1)

            total += labels.size(0)

            correct += (predicted == labels).sum().item()

    print(f"\nTest Accuracy: {100 * correct / total:.2f}%")
```

[ ]:
```
plt.figure(figsize=(10, 5))
plot_loss(losses, batches)
```

[ ]:
```
evaluate_model(model1, validation_loader)
```

Batch Normalization clearly has better results