
An implementation of https://arxiv.org/pdf/1512.03385.pdf (https://arxiv.org/pdf/1512.03385.pdf)

See section 4.2 for the model architecture on CIFAR-10

Some part of the code was referenced from below

https://github.com/pytorch/vision/blob/master/torc (https://github.com/pytorch/vision/blob/master/tor

---

In [1]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
```

In [7]:
```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
num_epochs = 5
learning_rate = 0.001

# Image preprocessing modules
transform = transforms.Compose([
    transforms.Pad(4),
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32),
    transforms.ToTensor()])
```

In [8]:
```
# CIFAR-10 dataset
train_dataset = torchvision.datasets.CIFAR10(root='../../data/',
                                             train=True,
                                             transform=transform,
                                             download=True)

test_dataset = torchvision.datasets.CIFAR10(root='../../data/',
                                            train=False,
                                            transform=transforms.ToTensor())

# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=100,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=100,
                                          shuffle=False)
```

```
Files already downloaded and verified
```

In [9]:
```
# 3x3 convolution
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                     stride=stride, padding=1, bias=False)
```

In [10]:
```
# Residual block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()
        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out
```

In [11]:
```
# ResNet
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.conv = conv3x3(3, 16)
        self.bn = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, layers[0])
        self.layer2 = self.make_layer(block, 32, layers[1], 2)
        self.layer3 = self.make_layer(block, 64, layers[2], 2)
        self.avg_pool = nn.AvgPool2d(8)
        self.fc = nn.Linear(64, num_classes)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if (stride != 1) or (self.in_channels != out_channels):
            downsample = nn.Sequential(
                conv3x3(self.in_channels, out_channels, stride=stride),
                nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(block(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out
```

In [12]:
```
model = ResNet(ResidualBlock, [2, 2, 2]).to(device)


# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

In [13]:
```
# For updating learning rate
def update_lr(optimizer, lr):
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr

# Train the model
total_step = len(train_loader)
curr_lr = learning_rate
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print ("Epoch [{}/{}], Step [{}/{}] Loss: {:.4f}"
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))

    # Decay learning rate
    if (epoch+1) % 20 == 0:
        curr_lr /= 3
        update_lr(optimizer, curr_lr)
```

```
Epoch [1/5], Step [100/500] Loss: 1.8060
Epoch [1/5], Step [200/500] Loss: 1.3315
Epoch [1/5], Step [300/500] Loss: 1.0736
Epoch [1/5], Step [400/500] Loss: 1.2652
Epoch [1/5], Step [500/500] Loss: 1.2130
Epoch [2/5], Step [100/500] Loss: 0.9698
Epoch [2/5], Step [200/500] Loss: 1.0678
Epoch [2/5], Step [300/500] Loss: 1.1806
Epoch [2/5], Step [400/500] Loss: 0.9964
Epoch [2/5], Step [500/500] Loss: 0.7356
Epoch [3/5], Step [100/500] Loss: 0.7760
Epoch [3/5], Step [200/500] Loss: 0.9168
Epoch [3/5], Step [300/500] Loss: 0.9050
Epoch [3/5], Step [400/500] Loss: 0.7955
Epoch [3/5], Step [500/500] Loss: 0.7463
Epoch [4/5], Step [100/500] Loss: 0.8532
Epoch [4/5], Step [200/500] Loss: 0.8871
Epoch [4/5], Step [300/500] Loss: 0.7609
Epoch [4/5], Step [400/500] Loss: 0.7302
Epoch [4/5], Step [500/500] Loss: 0.6396
Epoch [5/5], Step [100/500] Loss: 0.6814
Epoch [5/5], Step [200/500] Loss: 0.6683
Epoch [5/5], Step [300/500] Loss: 0.7848
Epoch [5/5], Step [400/500] Loss: 0.6743
Epoch [5/5], Step [500/500] Loss: 0.6952
```

In [14]:
```
# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Accuracy of the model on the test images: {} %'.format(100 * correct /
```

```
Accuracy of the model on the test images: 74.27 %
```

In [ ]:
```

```