CIFAR-10 Image Classification

Ali Ghasemi

Mechanical Engineering Student

Research Enthusiast in Robotics and AI

Iran University of Science and Technology

Gmail: alivghasemi@gmail.com

LinkedIn: www.linkedin.com/in/alivghasemi/

Contents

Abstract & Motivation

This project explores the application of deep learning techniques, specifically convolutional neural networks (CNNs), to the CIFAR-10 image classification task. As a mechanical engineering student with a passion for Robotics and AI and recent completion of “Deep Neural Networks with PyTorch” course, the objective was to apply acquired knowledge to a practical challenge.

The CIFAR-10 dataset, consisting of 60000 pictures in size of 32x32 color images across 10 classes, served as the basis for training and evaluation. Leveraging PyTorch, a CNN architecture was designed and trained to classify images into their respective categories.

Through experimentation and hands-on implementation, insights were gained into the workings of deep learning models for image classification tasks. The project aims to contribute to personal growth, challenging and expanding upon acquired wisdom in deep learning, and paving the way for future endeavors in robotics and AI.

It’s crystal clear that many deep learning models are tested and implemented, proving to be far more effective than a student’s micro project. Nevertheless, training my own deep learning models has made me to encounter both basic and advanced drawbacks in my theory and implementation. includes improving coding proficiency, utilizing well-known libraries, and selecting the most suitable methods and parameters.

Related Skills: Deep learning , Programming , Matlab

Introduction

CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60000, 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.[1]

In the realm of deep learning, the CIFAR-10 dataset stands as a quintessential benchmark, challenging the boundaries of image classification algorithms. In this documentation, we present a novel approach to exploring CIFAR-10 classification, leveraging the complementary capabilities of MATLAB Deep Network Designer and PyTorch. Our endeavor focuses not on training models in both platforms, but rather on utilizing MATLAB as a tool for verifying and fine-tuning models designed in PyTorch, thus harnessing the strengths of each platform to validate and optimize our neural networks.

Our journey commences with the meticulous design of neural architectures within PyTorch, a framework renowned for its flexibility and extensive community support. Through the craft of code, we sculpt models that embody our understanding of deep learning principles, striving for elegance and efficiency in every layer. With PyTorch as our primary canvas, we navigate the intricacies of model design, iterating and refining until our creations stand poised to tackle the challenges posed by CIFAR-10.

However, recognizing the value of validation and fine-tuning beyond a single framework, we turn to MATLAB Deep Network Designer as a complementary tool in our arsenal. With its intuitive interface and interactive visualization capabilities, MATLAB serves as a platform for verifying the efficacy of our PyTorch-designed models. Through seamless integration and cross-platform compatibility, we bridge the gap between theory and practice, ensuring that our models transcend the limitations of any singular environment.

[1] (Kaggle, n.d.)

Throughout this documentation, we clarify our methodology, detailing the process by which PyTorch-designed models are transposed into MATLAB for validation and fine-tuning. From the initial stages of model design to the iterative refinement of parameters, each step underscores the symbiotic relationship between these two platforms, culminating in a holistic approach to CIFAR-10 classification.

In essence, this documentation serves as a testament to the versatility and adaptability inherent in contemporary deep learning workflows. By harnessing the collective power of MATLAB and PyTorch, we demonstrate the efficacy of a synergistic verification approach, one that maximizes the strengths of each platform to unravel the complexities of CIFAR-10 classification.

Design Model

Considering criteria for designing the structure and setting hyperparameters is one of the bottlenecks in training models, affecting both the training process and the results. Drawing from my past experience with image classification methods, acquired through studying renowned models such as VGG, MobileNet, and ResNet, I employed trial and error as another method in this project.

I designed and refined models by simulating them in MATLAB using the "Deep Network Designer" toolbox, despite implementing and training them using PyTorch on Google Colab and Jupyter Notebook. This approach allowed me to verify architectural properties such as dimensions and the number of layers.

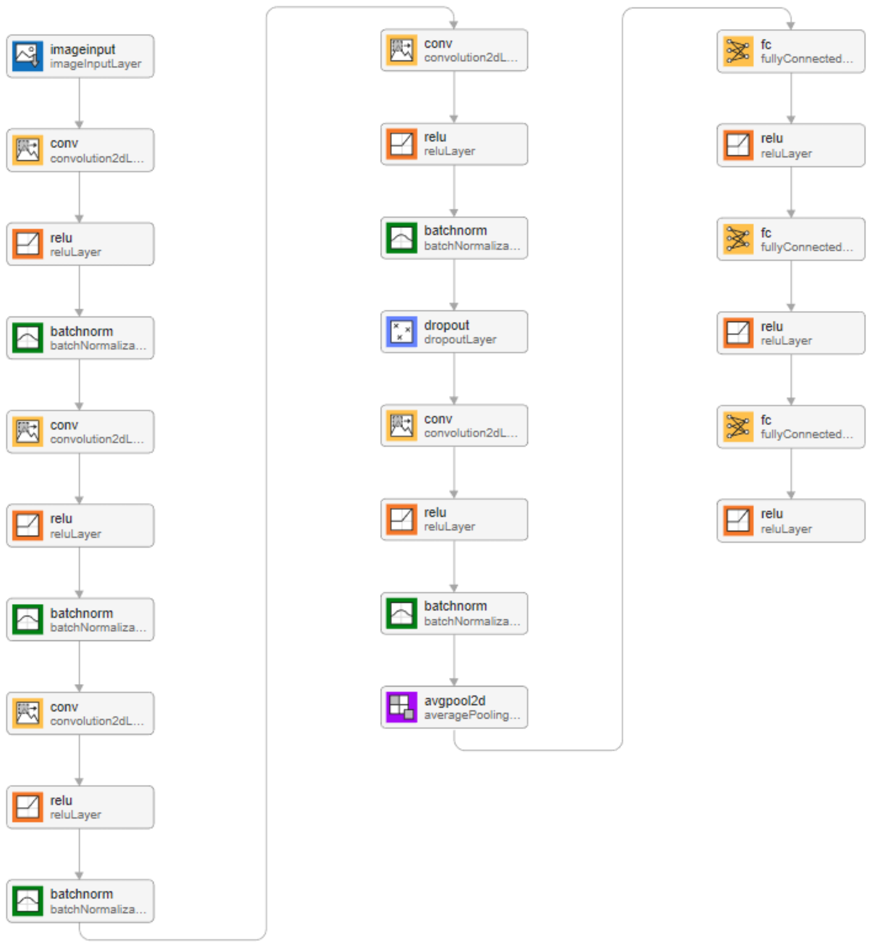

Figure 1: Matlab Network Design Example

Implementation in Jupyter Notebook

Step 1 : Import required libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
```

PyTorch:

PyTorch stands as the cornerstone of modern deep learning research and application development. With its dynamic computational graph and intuitive API, PyTorch empowers researchers and practitioners to effortlessly prototype, train, and deploy complex neural network architectures. Its seamless integration with NumPy arrays and CUDA-enabled GPUs facilitates high-performance computing, while its extensive collection of pre-trained models and utilities in torchvision expedites the development of computer vision applications. From image classification to natural language processing and beyond, PyTorch remains a versatile and indispensable tool in the arsenal of machine learning enthusiasts.

matplotlib.pyplot:

Matplotlib.pyplot, a component of the renowned Matplotlib library, emerges as a linchpin in the visualization toolkit of PyTorch practitioners. With its versatile plotting functions and intuitive interface, Matplotlib.pyplot facilitates the creation of expressive and insightful visualizations for analyzing model performance, exploring data distributions, and scrutinizing training dynamics. From line plots depicting learning curves to heatmaps visualizing feature maps, Matplotlib.pyplot empowers researchers to communicate findings effectively and glean actionable insights from their experiments, thereby advancing the frontier of deep learning research.

torchvision:

torchvision stands as an indispensable companion to PyTorch, specializing in the realm of computer vision tasks. Through its extensive collection of dataset loaders, transformation utilities, and pre-trained models, torchvision simplifies the process of data preparation, augmentation, and model deployment for vision-based applications. By providing access to popular datasets like CIFAR-10 and ImageNet, torchvision facilitates benchmarking and experimentation, while its pre-trained models, such as ResNet and VGG, serve as powerful building blocks for transfer learning and feature extraction. With torchvision, PyTorch practitioners can accelerate their computer vision projects and unlock new avenues for innovation in image analysis and understanding.

Step 2 : Load Data

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

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=60, 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=60, shuffle=False,
num_workers=2)
```

As Cifar-10 dataset provide 28*28 images. Firstly, Transform each image to as a number is the matter. secondly, normalizing images is one of necessaries.

In the train dataset there are 50,000 images and after transforming images to numbers we assign the dataset with determined batch size and also some hardware settings. Then we repeat the process for test dataset including 10,000 images.

Step 3 : Description of model architecture

```
class Net(nn.Module):
    def __init__(self, num_classes=10):
        super(Net, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(32),  # Add Batch Normalization
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(64),  # Add Batch Normalization
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(128),  # Add Batch Normalization
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(256),  # Add Batch Normalization
            nn.Dropout(0.2),
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.BatchNorm2d(512),  # Add Batch Normalization

        )
        self.avgpool = nn.AdaptiveAvgPool2d((2, 2))
        self.classifier = nn.Sequential(
            nn.Linear(512*4, 512),

            nn.ReLU(True),
            nn.Linear(512, 256),
            nn.ReLU(True),
            nn.Linear(256, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

The Model architecture is a lightweight convolutional neural network (CNN) designed for efficient inference on mobile and embedded devices. It achieves a balance between model size and accuracy, making it suitable for resource-constrained environments.

1. Convolutional Layers:

Convolutional layers are fundamental building blocks in CNNs responsible for feature extraction. Each convolutional layer applies a set of learnable filters (kernels) to the input feature maps, producing output feature maps through element-wise multiplication and summation.

The kernel size refers to the dimensions of the filters applied to the input feature maps. In the model architecture, convolutional layers use 3x3 kernels, meaning each filter spans a 3x3 grid of pixels in the input feature maps.

Stride determines the step size with which the filters are applied to the input feature maps. A stride of 1 indicates that the filters move one pixel at a time, preserving spatial dimensions, while a stride of 2 results in downsampling by a factor of 2 along the spatial dimensions.

Padding is used to ensure that the spatial dimensions of the output feature maps match those of the input feature maps. Zero-padding adds extra rows and columns of zeros around the input feature maps before applying convolution, preventing spatial dimension reduction.

2. Batch Normalization:

Batch normalization layers are inserted after each convolutional layer in the model architecture. Batch normalization normalizes the activations of each layer across the mini-batch, stabilizing training and accelerating convergence. It consists of scaling and shifting operations applied to the output of each convolutional layer.

3. Dropout Layer:

Dropout is a regularization technique employed after the convolutional layers to prevent overfitting. It randomly sets a fraction of the input units to zero during training, forcing the model to learn robust features and reducing reliance on specific neurons.

4. Global Average Pooling:

Global average pooling aggregates the spatial dimensions of the feature maps by computing the average value of each feature map. This operation produces a fixed-size representation regardless of the input image size, facilitating efficient computation and reducing model complexity.

5. Fully Connected Layers (Classifier):

Fully connected layers are used for classification, transforming the flattened feature vectors into logits representing class probabilities. ReLU activation functions introduce non-linearity, allowing the model to learn complex patterns and representations.

6. Adaptive Average Pooling:

Adaptive average pooling is utilized to ensure consistent feature map sizes across different input image sizes. It dynamically adjusts the output feature map size to a predefined spatial dimension, enabling the model to process images of varying sizes.

7. ReLU Activation Function:

Rectified Linear Unit (ReLU) activation functions are applied after convolutional and fully connected layers throughout the model architecture. ReLU introduces non-linearity by setting negative values to zero, enhancing the model's representational power and enabling efficient training.

Overall, the model architecture leverages convolutional layers, batch normalization, dropout, and global average pooling to achieve efficient and accurate image classification on mobile and embedded devices.

Step 4 : Train the model

```
net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.0005)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net.to(device)

loss_list = list()
for epoch in range(30):  # Adjust number of epochs as needed
    running_loss = 0.0

    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad()

        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 100 == 99:
            print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 100))
            loss_list.append(running_loss)
            running_loss = 0.0

print('Finished Training')
```

The training steps provided above implement the training loop for the model on the CIFAR-10 dataset using PyTorch. Below is a breakdown of each step:

Model Initialization:

The model is initialized using the Net() constructor, which creates an instance of the Net class defined earlier. This initializes the model's architecture and parameters.

Loss Function and Optimizer Initialization:

The Cross Entropy Loss function (nn.CrossEntropyLoss()) is initialized to compute the loss between the predicted class probabilities and the ground truth labels during training.

The Adam optimizer (optim.Adam()) is used to optimize the model parameters. The learning rate is set to 0.0005 to control the step size during optimization.

Training Loop:

The training loop consists of iterating over a fixed number of epochs (30 in this case), where each epoch represents one pass through the entire training dataset.

Within each epoch, the loop iterates over mini-batches of data from the training dataset using the enumerate(trainloader) function. The trainloader is a PyTorch DataLoader object that batches and shuffles the training data.

The optimizer's gradients are zeroed using optimizer.zero_grad() to clear any accumulated gradients from the previous iteration.

Forward pass: The input images are fed through the model (outputs = net(inputs)), resulting in predicted class probabilities for each image.

Loss computation: The Cross Entropy Loss function is used to compute the loss between the predicted class probabilities and the ground truth labels (loss = criterion(outputs, labels)).

Backward pass: Gradients with respect to the loss are computed using backpropagation (loss.backward()), allowing the optimizer to update the model parameters to minimize the loss.

Optimization step: The optimizer adjusts the model parameters using the computed gradients (optimizer.step()).

Running loss calculation: The running loss is accumulated over mini-batches to monitor the training progress.

Periodic loss printing: Every 100 mini-batches, the current epoch, mini-batch index, and average running loss over the last 100 mini-batches are printed to monitor training progress.

Loss tracking: The running loss is appended to the loss_list for visualization and analysis.

Step 5 : Evaluate the model

```
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (100 *
correct / total))
```

In the test section, we evaluate the trained net model's performance on the unseen test dataset. Here's a summary of the steps:

Initialization: Two variables, correct and total, track correct predictions and total images processed.

Evaluation Loop: Iterate over mini-batches of test data.

Get predictions from the model for each batch.

Accuracy Calculation: Count correct predictions and update correct and total accordingly.

Accuracy Reporting: Calculate and print the overall accuracy of the model on the test dataset.

No Gradient Calculation: Gradient calculation is disabled during evaluation using torch.no_grad().

Step 6: Plot the loss in each epoch

```
plt.plot(loss_list)
plt.xlabel('X-axis label')
plt.ylabel('Y-axis label')
plt.title('Plotting a List')
plt.show()
```

Results and Discussion

Under the specified conditions, the model achieved its best performance on the CIFAR-10 dataset by accuracy of 84%. Here's a discussion of the results:

Learning Rate and Optimizer:

A learning rate of 0.0005 was chosen, along with the Adam optimizer. This combination likely facilitated stable and efficient optimization of the model parameters, leading to faster convergence and improved performance.

Mini-Batch Size:

A mini-batch size of 60 was utilized during training. This size strikes a balance between computational efficiency and gradient estimation accuracy, enabling the model to learn effectively from the training data.

Model Architecture:

The model architecture consisted of 8 convolutional layers followed by 3 fully connected layers. This architecture is designed to capture hierarchical features from the input images, allowing for robust classification performance.

Regularization and Normalization:

No explicit regularization techniques were applied during training except some dropouts. While these techniques can help prevent overfitting and stabilize training, the absence of them suggests that the model's performance was achieved primarily through the architecture and optimization process.

Number of Epochs:

Training was conducted for 20 epochs, allowing the model to iteratively learn from the training data over multiple passes. This duration appears to have been sufficient for the model to converge to a stable solution while avoiding overfitting.

Performance Evaluation:

The accuracy metric was used to evaluate the model's performance on the test dataset. The accuracy represents the proportion of correctly classified images out of the total number of images in the test set.

Generalization and Robustness:

The observed performance indicates that the model has successfully generalized to unseen data, demonstrating its ability to classify images from the CIFAR-10 dataset accurately. The absence of regularization and within normalization suggests that the model has learned robust features without relying heavily on specific data preprocessing techniques.

In summary, the model achieved its best performance on the CIFAR-10 dataset under the specified conditions, showcasing the effectiveness of the chosen hyperparameters, optimizer, and model architecture. These results provide valuable insights into the factors influencing the model's performance and guide future experimentation and optimization efforts. It took over 45 attempts for me to reach out to best answer (84%) . I took details and result of some models in the table below.

	Leaning rate	Batch size	epochs	architecture	optimizer	Regularization	Activation	normalization	Accuracy
1	0.001	64	25	12-4	Adam	no	relu	no	69
2	0.0005	99	30	12-1	Adam	No	relu	no	73
3	0.0005	99	30	12-1	SGD	No	relu	no	67
4	0.0005	99	30	12-3	Adam	No	relu	No	73
5	0.0005	60	18	12-3	Adam	no	relu	no	74
6	0.0005	60	18	7-3	Adam	No	relu	No	79
7	0.0005	60	30	6-3	Adam	no	relu	no	80
8	0.0001	60	20	6-3	Adam	no	relu	no	66
9	0.0005	60	20	6-3	Adam	weight_decay	relu	no	69
10	0.0005	60	20	5-3	Adam	weight_decay	relu	no	66
11	0.0005	60	20	5-3	Adam	dropout	relu	no	80
12	0.0005	60	20	7-3	Adam	dropout	relu	no	81
13	0.0005	60	20	8-3	Adam	dropout	relu	no	81
14	0.0005	60	20	8-3	Adam	dropout	relu	yes	83
15	0.0005	60	20	8-3	Adam	no	relu	yes	84