Deep Learning: Mini Project Report

Chenxi Liu, Jalil Douglas, Dailin Ji

New York University, Tandon School of Engineering
https://github.com/TigaJi/DL-mini-project

Overview

In this project, our goal is to develop a lightweight version of ResNet (Zhang, Ren and Sun 2015) for image classification tasks. We achieved this by modifying the residual block within ResNet, resulting in a significant reduction in the number of parameters without sacrificing much of the model's capability. Additionally, we employed various preprocessing techniques, such as normalization and data augmentation, to enhance performance and achieve faster convergence. Finally, we experimented with multiple combinations of model parameters and optimizer to obtain an optimal model. As a result, we achieved a prediction accuracy of 91.47% on the CIFAR-10 (Krizhevsky and Hinton 2009) dataset with approximately 2.8 million trainable parameters. The codebase could be found through this publicly accessible github repository (https://github.com/TigaJi/DL-mini-project). (Ji, Liu and Douglas 2023)

Methodology

The architecture of this work consulted the github repository by Mountchicken (Mountchicken 2021) and class demo repository (Garimella 2023)

Datasets

In this project, the training dataset and testing dataset use the predetermined datasets by the CIFAR10 creators, which contains 50,000 training images and 10,000 test images.

Data Preprocessing

In order to help the model to learn features that are more invariant to the overall pixel values distribution and scale, we firstly calculate the mean and standard deviation of the training set data then apply the normalization function for each channel, this can help improve the convergence and overall performance of the neural network during training. Before applying the normalization, we also performed some data augmentation to artificially increase the diversity of the dataset. This could help with generating more training examples and mitigate potential overfitting. We performed random rotation, random horizontal flip, random crop, color alteration for the data augmentation for the training dataset and kept the test dataset plain.

ResNet Architecture

The ResNet architecture used in this project consists of multiple layers, mainly convolutional (Conv2d), batch normalization (BatchNorm2d), ReLU activation, and modified residual blocks followed by a fully connected layer. The network starts with an input size of 32x32 and gradually reduces spatial dimensions while increasing the number of channels. The final output of the network is a 10-dimensional vector, which represents class probabilities for each input.

Each ModifiedResidualBlock consists of two convolutional layers (Conv2d) with kernel size 3, followed by batch normalization (BatchNorm2d) and ReLU activation functions. Then a shortcut connection is included to allow the input to bypass the two convolutional layers. The shortcut connection consists of a 1x1 convolution followed by batch normalization when the input and output channels differ or the stride is not 1. The forward function defines the flow of data through the block, applying the two convolutional layers and merging the output with the shortcut connection before applying the final ReLU activation. In our model that gave the best result, each residual layer contains two residual blocks.

The ModifiedResNet consists of a convolutional layer with batch normalization and ReLU activation followed by four layers of residual blocks in our best trial. An adaptive average pooling layer reduces the spatial dimensions to 1x1, and the output is then flattened and passed through a fully connected layer to produce the final class predictions.

Increase the ModifiedResNet kernel size from 3 (with padding = 1) to 5 (with padding = 2) improved the pace of learning process as well as the highest accuracy from 40% to 60% in our early stage study, thus we kept using kernel size of 5 for all later trials. The padding size is also changed to maintain the consistent spatial dimensions of the output.

In order to properly initialize the parameter, we used *nn.init.kaiming_normal_* and *nn.init.constant_* to initialize the weights and biases of the convolutional and normalization layers in the architecture. This is supposed to ensure that the model could learn efficiently.

The *_make_layer* function is used to create a sequence of

ModifiedResidualBlock instances and stack them to form a layer within the ModifiedResNet architecture.

Training Process

From our preliminary trials, we figured out that in our architecture, 70 epoch training cycles could sufficiently represent the overall performance of the model. All of our training and testing cycles reached plateaus within 70 epochs, thus we kept this value for all later testing models. The three key evaluation parameters, train losses, test accuracies, learning rates are initialized to record the learning progress.

In each epoch, the model is firstly set to train mode, and iterates through the training data via loading data through DataLoader. The optimizer's gradients are initially set to zero, and apply the current model to the input images. CrossEntropy loss function is used to compute the loss between the model's predictions and the true labels. The gradients are then calculated by backpropagation, and the optimizer updates the model's weights. The learning rate scheduler (used CosineAnnealingLR in our case) adjusts the learning rate after each step. We stored the loss and learning rate in the initialized array accordingly. In our test scenarios, we tried both Adaptive Moment Estimation (Adam) and Stochastic Gradient Descent (SGD) as the optimizer. The outcome of the two optimizers are summarized in the next section.

Evaluating Process

The model is then set to evaluation mode. The test accuracy is assessed using the test dataset. Each test data batch is used for making a prediction where the predicted class is determined by selecting the class with the highest probability. The number of correct predictions and total samples is updated, and the test accuracy is calculated as the ratio of correct predictions to total samples. The best model with the highest accuracy is saved into *best_model.pth*.

Results and Analysis

There are many hyperparameters that could be optimized for the model. In our scenarios, we tried altering the batch size (128, 256, 512), optimizer (Adam, SGD), learning rate (0.001, 0.01, 0.1), number of residual layers (4 or 5) and the number of residual blocks in each layer (2 or 3). The total number of trainable parameters are also changed along with these hyperparameters (2,799,146, 4,587,306, 4,367,786).

When trying to increase the output channel numbers in the ModifiedResidualBlock, we figured out that it could easily expand the trainable parameters over 5 million thus we finally keep the architecture with output channels $32 \rightarrow 64 \rightarrow 128 \rightarrow 256$ in the ModifiedResNet.

The combination of hyperparameters are summarized in Table.1, all of our models reached the testing accuracy over 80%, with 7 of the models reaching the highest testing accuracy over 90%. The testing accuracy progress along with the number of epochs are summarized in Figure 1 to Figure 9. In the trials that yielded over 90% highest accuracy, the Model 5 reached a highest accuracy of 91.47%.

In Model 1, 5, 7, 8, by keeping the testing batch size the same, a 128 training batch size overall results in a smoother learning curve and more stable ending plateau after 70 epochs. This could be due to the fact that a smaller training batch size will result in a more frequent weights update for the model per epoch. This more frequent update acts as a gradually fine-tuning process of the parameter updating that could give a smoother accuracy curve over the course of training. Furthermore, using a smaller training batch size could introduce more noise in the gradient estimates, which can sometimes help the model to escape local minima and find better solutions. This batch size is considered optimal since it does not significantly slow down the learning process. The four models with this batch size all reached 80% around 10 epochs.

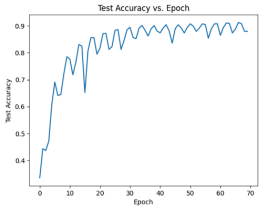


Figure 1: Model 0

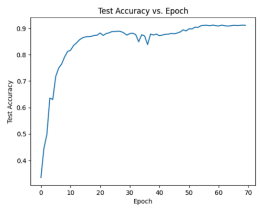


Figure 2: Model 1

The comparison of Model 0 and Model 2 (the only difference between these two models is the optimizer) shows that Adam is a better optimizer instead of SGD giving better accuracy of 91.28% instead of 83.25%. Therefore, we used SGD for the rest of the training.

Model 5, 6 and 7 show the impact of learning rate with all other parameters kept the same. The Model 6 shows that a learning rate of 0.1 is too slow for our scenario while Model 5 and Model 7 show competitive learning progress with Model 5 has a slightly faster learning progress and a slightly better final learning accuracy. The explanation could be that a larger learning rate, in theory, can speed up the learning process but may also result in overshooting the optimal solution thus causing oscillations around the optimal point and slow down the overall learning speed conversely.

Model 1 and Model 5 show the impact of the number of residual block layers. Model 1 contains one more layer with 3 residual blocks than Model 5, in which case, Model 1 has 60% more trainable parameters than Model 5 but did not yield a higher accuracy and a slightly slower learning speed. Therefore, we conclude that Model 5 is a better model with fewer trainable parameters that could save the computing power.

Model 5 and Model 8 show the impact of complexity of residual blocks, where Model 5 contains two residual blocks each residual layer while Model 8 contains three residual blocks each residual layer. The more complex architecture results in a near 60% increase in the total number of trainable parameters but did not improve the overall performance, therefore, we conclude that Model 5 is a better model with fewer trainable parameters but even slightly better final testing accuracy.

The total running times of each model are all very close on the scale of this project. In our testing environment (CoLab), the total cost of time for each trial is about 55min.

Model	Batch Size(training)	Batch Size(testing)	Optimizer	lr	#residual layer	#epoch	#params	Acc(%)
0	512	512	Adam	0.01	5	70	4,587,306	91.28
1	128	128	Adam	0.01	5	70	4,587,306	91.23
2	512	512	SGD	0.01	5	70	4,587,306	83.25
3	256	256	Adam	0.01	5	70	4,587,306	90.97
4	256	128	Adam	0.01	5	70	4,587,306	90.90
5	**128**	**128**	**Adam**	**0.01**	**4**	**70**	**2,799,146**	**91.47**
6	128	128	Adam	0.1	5	70	4,587,306	88.87
7	128	128	Adam	0.001	5	70	4,587,306	91.14
8	128	128	Adam	0.01	4	70	4,367,786	91.24

Table 1: Hyperparameters used for training process and results

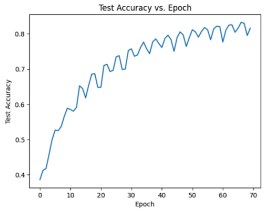


Figure 3: Model 2

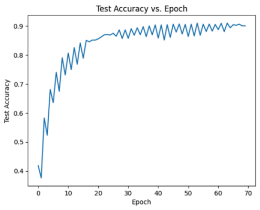


Figure 4: Model 3

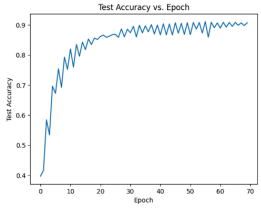


Figure 5: Model 4

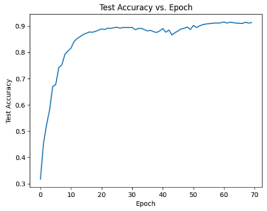


Figure 6: Model 5

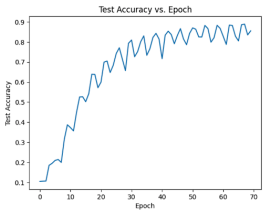


Figure 7: Model 6

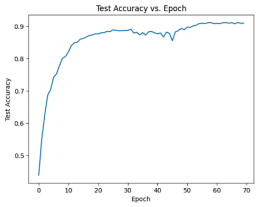


Figure 8: Model 7

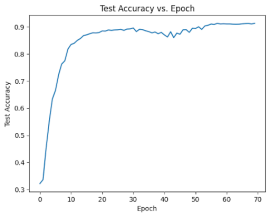


Figure 9: Model 8

Conclusion

Overall, the best model is Model 5 with training batch size of 128, Adam optimizer, learning rate of 0.01, four residual layers with two residual blocks each layer. Model 5 contains a total trainable parameter of 2,799,146 and yielding final testing accuracy of 91.47%. This outcome shows that it is not necessary that a more complicated model could yield a better performance.

For the future work of this project, we could increase trials to stress the stability of each model; Hyperparameter such as filter size, pool size and different scheduler could be studied. The learning rate could also be studied with finer increments along with the combinations of different optimizers since it could be one of the most impactive hyperparameters.

References

Garimella, K. V. 2023. Demo 05 - Implement ResNet and Visualize Loss Landscape. https://github.com/kvgarimella/dl-demos. Accessed: 2023-04-13.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Deep residual learning for image recognition. arXiv:1512.03385.

Ji, D.; Liu, C.; and Douglas, J. 2023. DL-mini-project. https://github.com/TigaJi/DL-mini-project. Accessed: 2023-04-13.

Krizhevsky, A.; and Hinton, G. 2009. Learning multiple layers of features from tiny images. Technical report, University of Toronto.

Mountchicken. 2021. ResNet18-CIFAR10. https://github.com/Mountchicken/ResNet18-CIFAR10. Accessed: 2023-04-13.

Please find this codebase through https://github.com/TigaJi/DL-mini-project/blob/main/Model_5_final.ipynb

```
#install torchvision
pip install torchvision
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: torchvision in /usr/local/lib/python3.9/dist-packages (0.15.1+cu118)
Requirement already satisfied: torch==2.0.0 in /usr/local/lib/python3.9/dist-packages (from torchvision) (2.0.0+cu118)
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages (from torchvision) (1.22.4)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.9/dist-packages (from torchvision) (8.4.0)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from torchvision) (2.27.1)
Requirement already satisfied: triton==2.0.0 in /usr/local/lib/python3.9/dist-packages (from torch==2.0.0->torchvision) (2.0.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.9/dist-packages (from torch==2.0.0->torchvision) (3.11.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.9/dist-packages (from torch==2.0.0->torchvision) (1.11.1)
Requirement already satisfied: networkx in /usr/local/lib/python3.9/dist-packages (from torch==2.0.0->torchvision) (3.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.9/dist-packages (from torch==2.0.0->torchvision) (3.1.2)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.9/dist-packages (from torch==2.0.0->torchvision) (4.5.0)
Requirement already satisfied: lit in /usr/local/lib/python3.9/dist-packages (from triton==2.0.0->torch==2.0.0->torchvision) (16.0.1)
Requirement already satisfied: cmake in /usr/local/lib/python3.9/dist-packages (from triton==2.0.0->torch==2.0.0->torchvision) (3.25.2)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->torchvision) (2022.12.7)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests->torchvision) (3.4)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests->torchvision) (1.26.15)
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->torchvision) (2.0.12)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.9/dist-packages (from jinja2->torch==2.0.0->torchvision) (2.1.2)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.9/dist-packages (from sympy->torch==2.0.0->torchvision) (1.3.0)
```

```
#install torchsummary
pip install torchsummary
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: torchsummary in /usr/local/lib/python3.9/dist-packages (1.5.1)
```

```
#import torch modules
import torch
import torchvision
import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10
from torch.utils.data import DataLoader
```

calculate the mean and standard deviation values, which then can use in the transforms.Normalize function.

```
import numpy as np
import torchvision.datasets as datasets

# The first line of code loads the CIFAR-10 training dataset and stores it in the cifar10_train variable.
#The root parameter specifies the location where the dataset will be stored, train=True specifies that we
#want to load the training data, and download=True specifies that we want to download the dataset if it hasn't already been downloaded.

#The second line of code uses a list comprehension to convert each image in the cifar10_train dataset to a numpy array and then stacks
#them into a single large numpy array using the np.stack function. The resulting array is stored in the images variable.

#The third and fourth lines of code calculate the mean and standard deviation for each channel of the images in the images array.
#The mean variable is calculated by taking the mean of the images array along the 0th, 1st, and 2nd axes (which correspond to the color channels)
#and dividing by 255 to normalize the values between 0 and 1. Similarly, the std variable is calculated by taking the standard deviation
#of the images array along the same axes and dividing by 255.
```

```
# Load the CIFAR-10 training dataset
cifar10_train = datasets.CIFAR10(root='./data', train=True, download=True)

# Stack all the images into a single large numpy array
images = np.stack([np.array(image) for image, _ in cifar10_train])

# Calculate the mean and standard deviation for each channel
mean = images.mean(axis=(0, 1, 2)) / 255
std = images.std(axis=(0, 1, 2)) / 255

print("Mean:", mean)
print("Standard Deviation:", std)
```

```
Files already downloaded and verified
Mean: [0.49139968 0.48215841 0.44653091]
Standard Deviation: [0.24703223 0.24348513 0.26158784]
```

By normalizing the input data, we ensure that the values lie in a similar range, which helps the neural network learn the features more effectively. This is particularly important when working with images, as the pixel values can vary greatly depending on the content and lighting conditions. Normalizing the data helps mitigate these differences and makes the training process more stable.

```
# The code is defining two sets of data transformations: one for training data (transform_train) and one for testing data (transform_test).
# The code uses the Compose function from the torchvision.transforms module to combine multiple transformations into a single transformation pipeline.

#The transform_train pipeline includes several data augmentation techniques that introduce random variations in the training images to help prevent overfitting
#and improve model generalization:

#RandomRotation(5) applies a random rotation of up to 5 degrees to the image.
#RandomHorizontalFlip(0.5) randomly flips the image horizontally with a 50% probability.
#RandomCrop(32, padding=4) randomly crops a 32x32 pixel section of the image with a 4-pixel padding around the edges.
#ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2) randomly changes the brightness, contrast, and saturation of an image.

# After the data augmentation transformations, the images are converted to PyTorch tensors using ToTensor().
#Finally, the Normalize() function normalizes the tensor image with mean and standard deviation values of (0.4914, 0.4821, 0.4465)
#and (0.2470, 0.2434, 0.2615) respectively. These values were precomputed using the training dataset, and are used to ensure that the
#pixel values of the images are in a suitable range for neural network training.

#The transform_test pipeline is simpler, as it only applies the ToTensor() and Normalize() transformations to the test data.
#This is because data augmentation techniques should not be applied to the test data, as we want to evaluate the model on the original, unmodified images.

# data augmentation part
transform_train = transforms.Compose([
    transforms.RandomRotation(5),
    transforms.RandomHorizontalFlip(0.5),
    transforms.RandomCrop(32, padding=4),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2), #randomly changes the brightness, contrast, and saturation of an image
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4821, 0.4465), (0.2470, 0.2434, 0.2615)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4821, 0.4465), (0.2470, 0.2434, 0.2615)),
])
```

```
#The above code creates two datasets, trainset and testset, by loading the CIFAR-10 dataset with the specified transformations.

#The trainset is created using the CIFAR10 class from the torchvision.datasets module. The root parameter specifies the directory where the
#dataset will be stored. The train parameter is set to True to indicate that we want to load the training set. The download parameter is set
#to True to download the dataset if it has not already been downloaded. Finally, the transform parameter is set to transform_train which is
#a pipeline of data augmentation and normalization transformations that will be applied to the images in the training set.

#The testset is created in a similar manner, but with train parameter set to False to indicate that we want to load the test set.
#The transform parameter is set to transform_test, which is a simpler pipeline that only converts the images to tensors and normalizes them.

#By creating trainset and testset in this way, we can easily iterate over the images in the datasets and apply the specified
#transformations on-the-fly during training and testing of the deep learning model.

trainset = CIFAR10(root='./data', train=True, download=True, transform=transform_train)
testset = CIFAR10(root='./data', train=False, download=True, transform=transform_test)
```

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

what is a good batch size to start with?

```
#The above code creates two DataLoader objects, train_loader and test_loader, which are used to load the trainset and testset datasets in
#batches during training and testing of the deep learning model.

#The DataLoader class is a built-in PyTorch class from the torch.utils.data module that provides an iterable over a dataset.
#It loads the data in batches, applies any specified transformations on-the-fly, and shuffles the data (if shuffle=True) to
#prevent any biases in the order of the data during training.

#The train_loader is created using the DataLoader class with trainset as the dataset to load, a batch_size of 128 to specify the
#number of samples per batch, and shuffle=True to randomly shuffle the order of the data during training.

#The test_loader is created in a similar way, but with trainset replaced by testset and shuffle=False since we do not want to shuffle the test data.

#By creating train_loader and test_loader in this way, we can easily iterate over the batches of the datasets during training and testing of the deep learning model.

train_loader = DataLoader(trainset, batch_size=128, shuffle=True)
test_loader = DataLoader(testset, batch_size=128, shuffle=False)
```

```
#This is a PyTorch implementation of a modified version of the ResNet architecture, specifically designed for the CIFAR-10 dataset.
#The modified ResNet includes four blocks of residual layers, each with a different number of layers. The input image is first
#processed by a convolutional layer with a kernel size of 5 and stride of 1, followed by batch normalization and ReLU activation.
#The output of the first convolutional layer is then passed through four blocks of residual layers, each of which has a different number of layers.
#Each residual layer is implemented as a ModifiedResidualBlock module, which includes two convolutional layers with batch normalization and ReLU activation,
#and a shortcut connection to ensure that the input and output dimensions match.

#After the four blocks of residual layers, the output is passed through a global average pooling layer to get a feature vector of size 256,
#and then through a fully connected layer to produce the final output logits of size 10 (corresponding to the 10 classes in the CIFAR-10 dataset).
#The neural network is initialized using the Kaiming normal initialization for the convolutional layers, and constant initialization for the batch normalization layers.

#The ModifiedResNet class takes in two arguments: block, which is the class defining the residual blocks (in this case, ModifiedResidualBlock),
#and num_blocks, which is a list of four integers indicating the number of layers in each block.
#By default, the num_classes argument is set to 10, but can be changed if desired.

import torch.nn as nn
```

```
from torch.optim import SGD
from torch.optim.lr_scheduler import CosineAnnealingLR
from torchsummary import summary

class ModifiedResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ModifiedResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = self.relu(out)
        return out

class ModifiedResNet(nn.Module): ## need some initialization control
    def __init__(self, block, num_blocks, num_classes=10):
        super(ModifiedResNet, self).__init__()
        self.in_channels = 32 ##change
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5, stride=1, padding=2, bias=False)##change
        self.bn1 = nn.BatchNorm2d(32)##change
        self.relu = nn.ReLU(inplace=True)

        ##change
        self.layer1 = self._make_layer(block, 32, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 64, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 128, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 256, num_blocks[3], stride=2)
        #self.layer5 = self._make_layer(block, 256, num_blocks[4], stride=2)
        self.gap = nn.AdaptiveAvgPool2d((1,1))
        self.fc = nn.Linear(256, num_classes) #a fully connected layer

        #ensure that the neural network is initialized properly before training
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def _make_layer(self, block, out_channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_channels, out_channels, stride))
            self.in_channels = out_channels
```

```
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        #out = self.layer5(out)
        out = self.gap(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out
```

```
#This is a valid implementation of a modified ResNet18 architecture using the ModifiedResidualBlock defined earlier.
#The network consists of 4 stages, each containing 2 residual blocks. The output of the last stage is passed through a
#global average pooling layer and then a fully connected layer to produce the final output.

#However, note that the last residual block in the original ResNet18 architecture has 3 blocks instead of 2, so the implementation here is slightly modified.

def ModifiedResNet18():
    #return ModifiedResNet(ModifiedResidualBlock, [2, 2, 2, 2, 3])
    return ModifiedResNet(ModifiedResidualBlock, [2, 2, 2, 2])
    ## this could be changed
```

```
#The CosineAnnealingLR scheduler reduces the learning rate using a cosine annealing schedule.
#The learning rate starts from the initial learning rate and decreases gradually to the minimum learning rate
#over a specified number of epochs (T_max) following a cosine curve. After reaching the minimum learning rate, it
#starts increasing again following another cosine curve, and the cycle repeats.

#In this case, the scheduler will reduce the learning rate of the Adam optimizer from 0.01 to 0 over 200
#epochs using a cosine curve. At the end of 200 epochs, the learning rate will be eta_min which is set to 0.

device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = ModifiedResNet18().to(device)

# Get the summary
input_shape = (3, 32, 32)  # (channels, height, width)
summary(model, input_shape, device=device)

##change
#optimizer = SGD(model.parameters(), lr=0.001, momentum=0.9, weight_decay=1e-4)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
##could change the scheduler
scheduler = CosineAnnealingLR(optimizer, T_max=200, eta_min=0)
#scheduler=torch.optim.lr_scheduler.MultiStepLR(optimizer,milestones=[15,25,30],gamma=0.1)
```

```
      BatchNorm2d-31           [-1, 64, 16, 16]             128
             ReLU-32           [-1, 64, 16, 16]               0
ModifiedResidualBlock-33           [-1, 64, 16, 16]               0
           Conv2d-34            [-1, 128, 8, 8]          73,728
      BatchNorm2d-35            [-1, 128, 8, 8]             256
             ReLU-36            [-1, 128, 8, 8]               0
           Conv2d-37            [-1, 128, 8, 8]         147,456
      BatchNorm2d-38            [-1, 128, 8, 8]             256
           Conv2d-39            [-1, 128, 8, 8]           8,192
      BatchNorm2d-40            [-1, 128, 8, 8]             256
             ReLU-41            [-1, 128, 8, 8]               0
ModifiedResidualBlock-42            [-1, 128, 8, 8]               0
           Conv2d-43            [-1, 128, 8, 8]         147,456
      BatchNorm2d-44            [-1, 128, 8, 8]             256
             ReLU-45            [-1, 128, 8, 8]               0
           Conv2d-46            [-1, 128, 8, 8]         147,456
      BatchNorm2d-47            [-1, 128, 8, 8]             256
             ReLU-48            [-1, 128, 8, 8]               0
ModifiedResidualBlock-49            [-1, 128, 8, 8]               0
           Conv2d-50            [-1, 256, 4, 4]         294,912
      BatchNorm2d-51            [-1, 256, 4, 4]             512
             ReLU-52            [-1, 256, 4, 4]               0
           Conv2d-53            [-1, 256, 4, 4]         589,824
      BatchNorm2d-54            [-1, 256, 4, 4]             512
           Conv2d-55            [-1, 256, 4, 4]          32,768
      BatchNorm2d-56            [-1, 256, 4, 4]             512
             ReLU-57            [-1, 256, 4, 4]               0
ModifiedResidualBlock-58            [-1, 256, 4, 4]               0
           Conv2d-59            [-1, 256, 4, 4]         589,824
      BatchNorm2d-60            [-1, 256, 4, 4]             512
             ReLU-61            [-1, 256, 4, 4]               0
           Conv2d-62            [-1, 256, 4, 4]         589,824
      BatchNorm2d-63            [-1, 256, 4, 4]             512
             ReLU-64            [-1, 256, 4, 4]               0
ModifiedResidualBlock-65            [-1, 256, 4, 4]               0
AdaptiveAvgPool2d-66            [-1, 256, 1, 1]               0
           Linear-67                  [-1, 10]           2,570
================================================================
Total params: 2,799,146
Trainable params: 2,799,146
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 7.75
Params size (MB): 10.68
Estimated Total Size (MB): 18.44
----------------------------------------------------------------
```

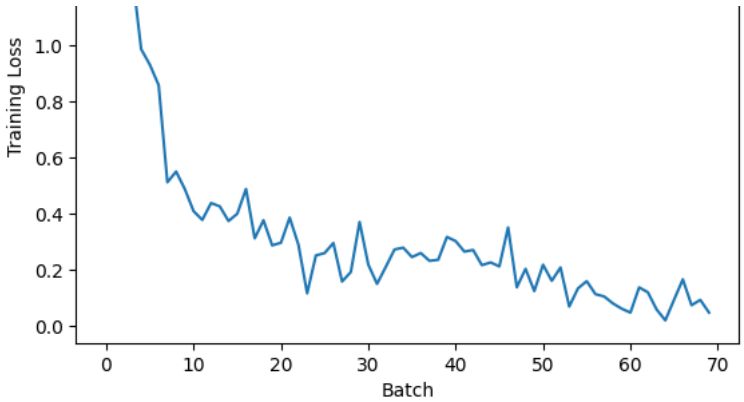
```
#This code trains a modified version of the ResNet18 model on the CIFAR-10 dataset. The training loop runs for num_epochs and consists of two parts:
#training and testing. During training, the model is put in training mode (model.train()) and the optimizer is zeroed (optimizer.zero_grad()).
#Then the model is used to compute the outputs and the loss using the cross-entropy loss function. The loss is backpropagated through the network
#and the optimizer is updated (optimizer.step()). The learning rate is also updated using a cosine annealing scheduler (scheduler.step()).

#After each epoch, the model is put in evaluation mode (model.eval()) and the test accuracy is evaluated by iterating through the test dataset
#and computing the accuracy as the number of correct predictions divided by the total number of samples. If the current accuracy is higher than
#the best accuracy seen so far, the model state is saved to disk (torch.save(model.state_dict(), save_path)).

#Finally, the training loss after each batch, the test accuracy after each epoch, and the learning rate after each epoch are all recorded in
#the train_losses, test_accuracies, and learning_rates lists, respectively. These can be used to visualize the training progress and diagnose any issues.

num_epochs = 70
best_accuracy = 0.0  # Initialize the best accuracy variable
```

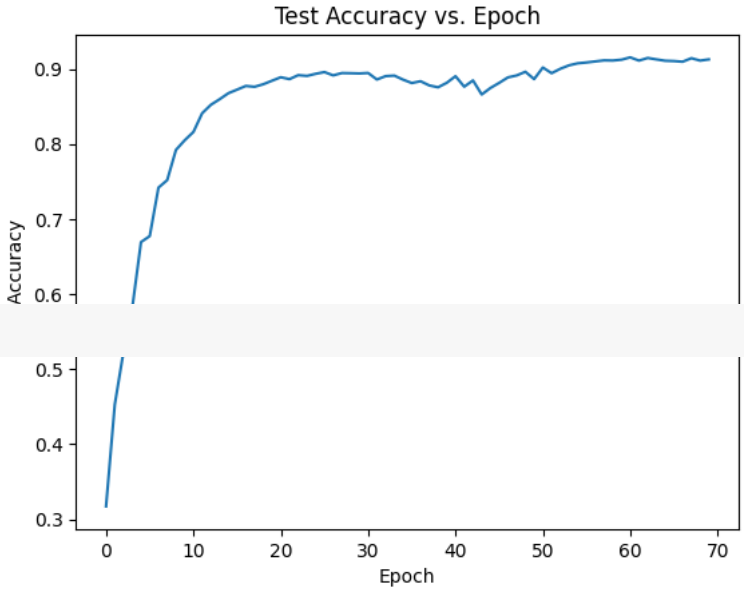
```
save_path = 'best_model.pth'  # Specify the path where the best model will be saved

train_losses = []
test_accuracies = []
learning_rates = []


for epoch in range(num_epochs):
    model.train()
    for i, (images, labels) in enumerate(train_loader):
      images, labels = images.to(device), labels.to(device)
      optimizer.zero_grad()
      outputs = model(images)
      loss = nn.CrossEntropyLoss()(outputs, labels)
      loss.backward()
      optimizer.step()
      scheduler.step()

    # Test accuracy evaluation
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_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()

    accuracy = correct / total
    print(f'Epoch [{epoch+1}/{num_epochs}], Accuracy: {accuracy:.4f}')
    # Save the best model
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        torch.save(model.state_dict(), save_path)
    # Append training loss after each batch
    train_losses.append(loss.item())

    # Append test accuracy after each epoch
    test_accuracies.append(accuracy)

    # Append learning rate after each epoch
    learning_rates.append(scheduler.get_last_lr()[0])
```

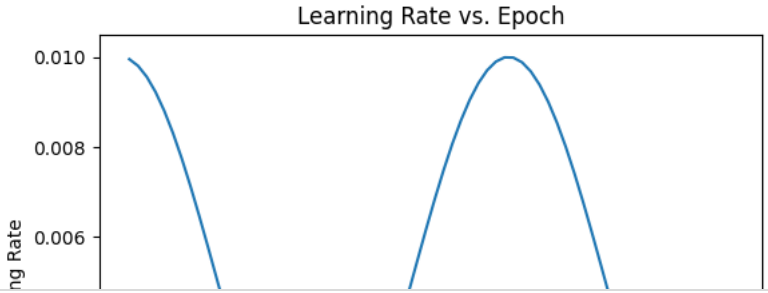
```
Epoch [25/70], Accuracy: 0.8936
Epoch [26/70], Accuracy: 0.8959
Epoch [27/70], Accuracy: 0.8915
Epoch [28/70], Accuracy: 0.8946
Epoch [29/70], Accuracy: 0.8944
Epoch [30/70], Accuracy: 0.8941
Epoch [31/70], Accuracy: 0.8947
Epoch [32/70], Accuracy: 0.8860
Epoch [33/70], Accuracy: 0.8905
Epoch [34/70], Accuracy: 0.8912
Epoch [35/70], Accuracy: 0.8858
Epoch [36/70], Accuracy: 0.8813
Epoch [37/70], Accuracy: 0.8837
Epoch [38/70], Accuracy: 0.8781
Epoch [39/70], Accuracy: 0.8756
Epoch [40/70], Accuracy: 0.8816
Epoch [41/70], Accuracy: 0.8904
Epoch [42/70], Accuracy: 0.8764
Epoch [43/70], Accuracy: 0.8848
Epoch [44/70], Accuracy: 0.8660
Epoch [45/70], Accuracy: 0.8747
Epoch [46/70], Accuracy: 0.8815
Epoch [47/70], Accuracy: 0.8888
Epoch [48/70], Accuracy: 0.8914
Epoch [49/70], Accuracy: 0.8963
Epoch [50/70], Accuracy: 0.8865
Epoch [51/70], Accuracy: 0.9020
Epoch [52/70], Accuracy: 0.8944
Epoch [53/70], Accuracy: 0.9003
Epoch [54/70], Accuracy: 0.9049
Epoch [55/70], Accuracy: 0.9076
Epoch [56/70], Accuracy: 0.9087
Epoch [57/70], Accuracy: 0.9101
Epoch [58/70], Accuracy: 0.9115
Epoch [59/70], Accuracy: 0.9113
Epoch [60/70], Accuracy: 0.9124
Epoch [61/70], Accuracy: 0.9155
Epoch [62/70], Accuracy: 0.9112
Epoch [63/70], Accuracy: 0.9147
Epoch [64/70], Accuracy: 0.9128
Epoch [65/70], Accuracy: 0.9110
Epoch [66/70], Accuracy: 0.9106
Epoch [67/70], Accuracy: 0.9097
Epoch [68/70], Accuracy: 0.9144
Epoch [69/70], Accuracy: 0.9112
Epoch [70/70], Accuracy: 0.9128
```

```
#These plots will help us to analyze the training process of our model. The first plot shows the training loss vs. the batch number.
#The second plot shows the test accuracy vs. the epoch number. The third plot shows the learning rate vs. the epoch number.

import matplotlib.pyplot as plt

# Plot training loss
plt.figure()
plt.plot(train_losses)
plt.xlabel('Batch')
plt.ylabel('Training Loss')
plt.title('Training Loss vs. Batch')

# Plot test accuracy
plt.figure()
plt.plot(test_accuracies)
```

```
plt.xlabel('Epoch')
plt.ylabel('Test Accuracy')
plt.title('Test Accuracy vs. Epoch')

# Plot learning rate
plt.figure()
plt.plot(learning_rates)
plt.xlabel('Epoch')
plt.ylabel('Learning Rate')
plt.title('Learning Rate vs. Epoch')

plt.show()
```


Training Loss
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
60
70
Batch


Test Accuracy vs. Epoch
Accuracy
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
10
20
30
40
50
60
70
Epoch


Learning Rate vs. Epoch
0.010
0.008
0.006