ECE-GY 6143
Deep Learning Mini Project

Raghav Rawat, Utkarsh Prakash Srivastava, Srushti Jagtap
rr3418, ups2006, sj4182
[1]Department of Electrical and Computer Engineering
Tandon School of Engineering,
New York University, NY, USA
Github Link : https://github.com/rawatraghav/ECE-GY-7123-DL-mini-project

Abstract

This project presents an enhanced ResNet architecture tailored for the CIFAR-10 dataset, integrating Basic Blocks and Squeeze-and-Excitation blocks to emphasize the significance of channel interdependencies. The proposed design achieves remarkable accuracy while adhering to a parameter-efficient methodology. It offers flexibility in layer configurations, kernel sizes, and dropout settings to optimize performance.

With a configuration comprising [4, 4, 3] blocks and an initial setup of 64 channels, this model embodies a strategic amalgamation of innovation and computational efficiency. It represents a notable advancement in the realm of resource-conscious deep learning solutions.

Introduction

The ResNet architecture revolutionized the construction of deeper neural networks with the introduction of residual blocks. The architecture variants, ranging from ResNet-18 to ResNet-152, demonstrated adaptability through increased depth, allowing deeper networks to learn more complex features without degradation.Each convolutional layer in a residual block is typically followed by batch normalization and a ReLU activation function. This standardizes the outputs of the convolution, stabilizing and speeding up the training. CIFAR-10 data [Krizhevsky and Hinton 2009] is loaded using a custom function that unpickles the dataset batches. CIFAR-10 consists of small images classified into 10 categories.

We have also used **schedulers** . OneCycleLR scheduler is designed to adjust the learning rate according to the "1cycle" policy, which is a technique that involves one cycle of learning rate adjustments over the training period.Cosine Annealing learning rate scheduler, which adjusts the learning rate following a cosine curve, decreasing initially and then increasing slightly towards the end of each cycle to facilitate better convergence. Various optimizers were compared, and it was determined that Stochastic Gradient Descent with momentum, was the most effective approach, as reported by [Liu, Gao, and Yin 2020].The final model satisfied the constraints and exhibited computational efficiency.

ResNet

The core idea behind ResNet is its use of residual blocks. These blocks allow the input to a set of layers to be added to their output, effectively enabling the training of very deep networks by alleviating the vanishing gradients problem. This is implemented using skip connections or shortcut connections that skip one or more layers.We utilize a customized Residual Neural Network (ResNet) architecture composed of three layers with 4, 4, and 3 *BasicBlock* units respectively. Each layer uses a 3x3 convolutional kernel size, complemented by 1x1 kernels in shortcut connections to maintain feature dimensionality. The network, uniform in its 64-channel configuration across all layers, concludes with an 8x8 kernel average pooling layer to reduce spatial dimensions efficiently. Dropout is not employed, and Squeeze-and-Excitation (SE) blocks are integrated to enhance feature recalibration capabilities. This architecture adheres to a strict parameter limit of under 5 million, ensuring the model's scalability and efficiency for robust image recognition tasks.

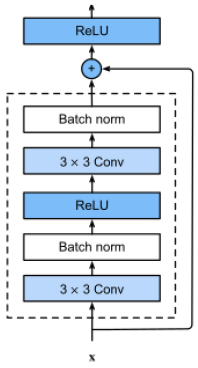


Figure 1: ResNet Block.

Methodology

PreProcessing

In our project, we performed data normalization with the mean values (0.4914, 0.4822, 0.4465) and standard deviation values (0.2023, 0.1994, 0.2010) are provided for both

training and test datasets. It involves adjusting the pixel values of the input images to have a mean and standard deviation that are closer to zero and one respectively. This normalization helps in stabilizing the training process and improving convergence. We utilized **data augmentation** to enhance the model's generalization capabilities, leveraging the `torchvision.transforms` module in PyTorch. Our data augmentation pipeline was defined using the `transforms.Compose()` function, comprising the following key transformations:

- Random Crop: Each image was randomly cropped to a size of 32x32 pixels, utilizing a padding of 4 pixels and `padding_mode='reflect'`.
- Random Horizontal Flip: Images were horizontally flipped with a 50% probability to introduce diversity in orientation.
- ToTensor Conversion: All images were converted to tensors, facilitating processing by the PyTorch model.
- Normalization: Pixel values were normalized based on predefined `data_statistics` to stabilize training by ensuring consistent input features as mentioned before.

Model

The architecture unfolded through a series of convolutional layers, commencing with 3x3 kernels that adeptly captured essential features from the input images. These layers were systematically integrated with batch normalization to ensure smooth and stable training, as the network progressed deeper through its numerous layers. Activation functions such as ReLU and Sigmoid were selected to introduce the necessary non-linearity, enabling the model to effectively tackle intricate patterns within the visual data. The proposed model can be visualized in figure 3.

A notable feature of this model was the inclusion of **Squeeze-and-Excitation blocks**, which were seamlessly woven into the convolutional framework. The network advanced from 64 feature maps in the initial layers to 256 in the deeper sections, preparing for a final classification over 10 distinct categories. This structure was well-aligned with the demands of the CIFAR-10 dataset, which is frequently used in machine learning research. The model's parameter count stood at 4,697,742, demonstrating its design efficiency and adherence to a self-imposed limit of 5 million parameters. This precision was further exemplified by the uniform distribution of 36,864 parameters within the BasicBlock layers, which illustrated a synergy between uniformity and functionality. The design journey culminated in a fully connected layer that housed 2,570 parameters, suitably dimensioned for delivering the classification results. The foundational layer of the model established a robust base with 1,728 parameters, paving the way for the sophisticated architecture that followed.

The architecture of the neural network model is summarized as follows:

- The initial convolutional layer, Conv2d-1, processed input into 64 feature maps, each 32x32.
- Batch normalization, via BatchNorm2d, followed, normalizing feature maps without changing their dimensions.
- The AdaptiveAvgPool2d layer reduced feature maps to a 1x1 dimension, facilitating feature consolidation.
- Squeeze-and-Excitation blocks, SEBlock, refined features channel-wise, preserving their shapes.
- With network deepening, as seen in BasicBlock-18 and BasicBlock-35, feature maps increased from 64 to 256 while spatial dimensions reduced for complexity management.
- Expansion in channel count in deeper layers signaled learning of more complex features.
- Finally, the network's output was through a Linear layer, aligned with CIFAR-10's ten-category classification.

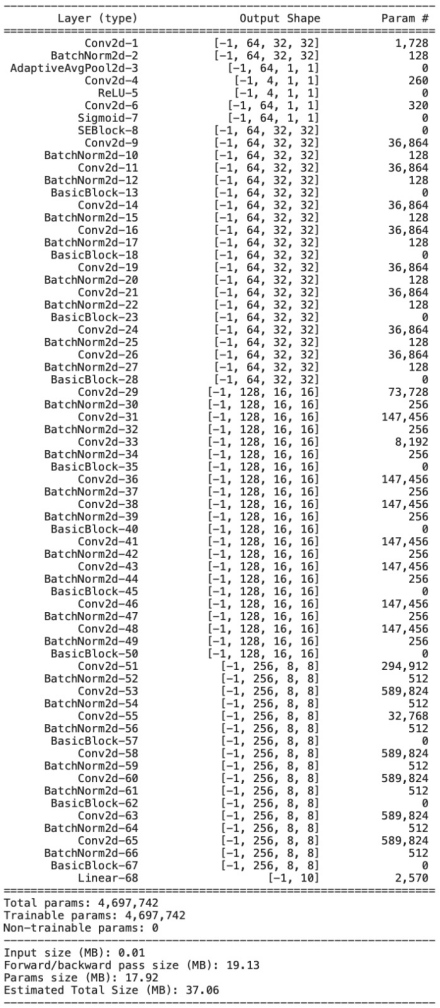
Model takes inspiration from **Pre-activation ResNet (Pre-Act ResNet)** [He et al. 2015]. It is a variation of the ResNet architecture designed to improve training stability and performance, particularly in deep neural networks. Pre-activation, ie, activation functions (e.g., ReLU) are applied before convolutional and batch normalization layers rather than after them. This helps alleviate the vanishing gradient problem and facilitates smoother gradient flow during training. It retains skip connections from the original ResNet architecture. These connections enable the network to learn residual functions, which are then added to the input of each block. This simplifies the learning process by focusing on residual features. Batch normalization is applied before the activation function in each residual block. This normalizes the activations, making training more stable and accelerating convergence. Typically, the network ends with a global average pooling layer followed by a fully connected layer with softmax activation for classification. This pooling operation reduces spatial dimensions to a single value per feature map, facilitating classification.

Hyperparameter Tuning

In our project, the ResNet model was configured with key hyperparameters for optimal training. We utilized convolutional kernel sizes of 3x3 and structured the network with three layers of residual blocks with 4, 4, and 3 blocks respectively, each layer featuring 64 channels. Shortcut connections utilized 1x1 kernels, and average pooling was conducted using an 8x8 kernel size.

We chose Stochastic Gradient Descent (SGD) as our **optimizer**, favored for its effectiveness in large-scale optimization problems. SGD was configured with a momentum of 0.9 to help navigate the loss landscape more smoothly and a learning rate (LR) starting at 0.1, adjusted dynamically with a CosineAnnealingLR scheduler to follow a cosine decay pattern. The training involved 200 epochs, a batch size of 128, and weight decay set at 0.0005 to prevent overfitting.

Additional mechanisms included Squeeze-and-Excitation blocks activated to enhance channel-wise feature recalibration. We did not employ dropout, opting instead for batch normalization to control overfitting. Gradient clipping was set at 0.1 to prevent gradient explosions, and 16 worker



Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 32, 32]	1,728
BatchNorm2d-2	[-1, 64, 32, 32]	128
AdaptiveAvgPool2d-3	[-1, 64, 1, 1]	0
Conv2d-4	[-1, 4, 1, 1]	260
ReLU-5	[-1, 4, 1, 1]	0
Conv2d-6	[-1, 64, 1, 1]	320
Sigmoid-7	[-1, 64, 1, 1]	0
SEBlock-8	[-1, 64, 32, 32]	0
Conv2d-9	[-1, 64, 32, 32]	36,864
BatchNorm2d-10	[-1, 64, 32, 32]	128
Conv2d-11	[-1, 64, 32, 32]	36,864
BatchNorm2d-12	[-1, 64, 32, 32]	128
BasicBlock-13	[-1, 64, 32, 32]	0
Conv2d-14	[-1, 64, 32, 32]	36,864
BatchNorm2d-15	[-1, 64, 32, 32]	128
Conv2d-16	[-1, 64, 32, 32]	36,864
BatchNorm2d-17	[-1, 64, 32, 32]	128
BasicBlock-18	[-1, 64, 32, 32]	0
Conv2d-19	[-1, 64, 32, 32]	36,864
BatchNorm2d-20	[-1, 64, 32, 32]	128
Conv2d-21	[-1, 64, 32, 32]	36,864
BatchNorm2d-22	[-1, 64, 32, 32]	128
BasicBlock-23	[-1, 64, 32, 32]	0
Conv2d-24	[-1, 64, 32, 32]	36,864
BatchNorm2d-25	[-1, 64, 32, 32]	128
Conv2d-26	[-1, 64, 32, 32]	36,864
BatchNorm2d-27	[-1, 64, 32, 32]	128
BasicBlock-28	[-1, 64, 32, 32]	0
Conv2d-29	[-1, 128, 16, 16]	73,728
BatchNorm2d-30	[-1, 128, 16, 16]	256
Conv2d-31	[-1, 128, 16, 16]	147,456
BatchNorm2d-32	[-1, 128, 16, 16]	256
Conv2d-33	[-1, 128, 16, 16]	8,192
BatchNorm2d-34	[-1, 128, 16, 16]	256
BasicBlock-35	[-1, 128, 16, 16]	0
Conv2d-36	[-1, 128, 16, 16]	147,456
BatchNorm2d-37	[-1, 128, 16, 16]	256
Conv2d-38	[-1, 128, 16, 16]	147,456
BatchNorm2d-39	[-1, 128, 16, 16]	256
BasicBlock-40	[-1, 128, 16, 16]	0
Conv2d-41	[-1, 128, 16, 16]	147,456
BatchNorm2d-42	[-1, 128, 16, 16]	256
Conv2d-43	[-1, 128, 16, 16]	147,456
BatchNorm2d-44	[-1, 128, 16, 16]	256
BasicBlock-45	[-1, 128, 16, 16]	0
Conv2d-46	[-1, 128, 16, 16]	147,456
BatchNorm2d-47	[-1, 128, 16, 16]	256
Conv2d-48	[-1, 128, 16, 16]	147,456
BatchNorm2d-49	[-1, 128, 16, 16]	256
BasicBlock-50	[-1, 128, 16, 16]	0
Conv2d-51	[-1, 256, 8, 8]	294,912
BatchNorm2d-52	[-1, 256, 8, 8]	512
Conv2d-53	[-1, 256, 8, 8]	589,824
BatchNorm2d-54	[-1, 256, 8, 8]	512
Conv2d-55	[-1, 256, 8, 8]	32,768
BatchNorm2d-56	[-1, 256, 8, 8]	512
BasicBlock-57	[-1, 256, 8, 8]	0
Conv2d-58	[-1, 256, 8, 8]	589,824
BatchNorm2d-59	[-1, 256, 8, 8]	512
Conv2d-60	[-1, 256, 8, 8]	589,824
BatchNorm2d-61	[-1, 256, 8, 8]	512
BasicBlock-62	[-1, 256, 8, 8]	0
Conv2d-63	[-1, 256, 8, 8]	589,824
BatchNorm2d-64	[-1, 256, 8, 8]	512
Conv2d-65	[-1, 256, 8, 8]	589,824
BatchNorm2d-66	[-1, 256, 8, 8]	512
BasicBlock-67	[-1, 256, 8, 8]	0
Linear-68	[-1, 10]	2,570

Total params: 4,697,742
Trainable params: 4,697,742
Non-trainable params: 0

Input size (MB): 0.01
Forward/backward pass size (MB): 19.13
Params size (MB): 17.92
Estimated Total Size (MB): 37.06

Figure 2: Transformation of Output shape at each Layer

threads were used to improve data loading efficiency. Data augmentation and normalization were both actively employed to enhance the model's generalization capabilities. Training initiated from scratch as indicated by a resume checkpoint flag set to '0'.

System Specification

NYU HPC VM
CPU: 8 Virtualized Cores of Intel Xeon-Platinum 8286
GPU: Nvidia Quadro RTX 8000
System Memory: 64 GB
Python Version: 3.9.0
CUDA version: v12.1
Torch Version: 2.2.2

Results

The model was trained on the CIFAR-10 dataset using PyTorch, achieving promising results on both the test and hidden datasets. During training, the model demonstrated a consistent improvement in performance, as evidenced by the increasing accuracy and decreasing loss over the epochs.

The training process involved multiple epochs, during which the model iteratively learned from the training data to improve its predictive capabilities. Each epoch involved processing batches of training samples, updating the model parameters using backpropagation, and evaluating the model's performance on the validation set.

After completing the training process, the model was evaluated on both the test and hidden datasets to assess its generalization performance. The test dataset yielded an accuracy of **95%**, indicating that the model performed well on unseen data from the same distribution as the training set.

Furthermore, the model was tested on a hidden dataset, achieving an accuracy of **85.5%**. This result suggests that the model's performance remains robust when applied to data from a different distribution, demonstrating its ability to generalize effectively.

Overall, the experimental results indicate that the trained model exhibits strong predictive performance on both the test and hidden datasets, underscoring its effectiveness in image recognition tasks.

References

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Deep Residual Learning for Image Recognition. *arXiv*.

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

Liu, Y.; Gao, Y.; and Yin, W. 2020. An improved analysis of stochastic gradient descent with momentum. *Advances in Neural Information Processing Systems*.

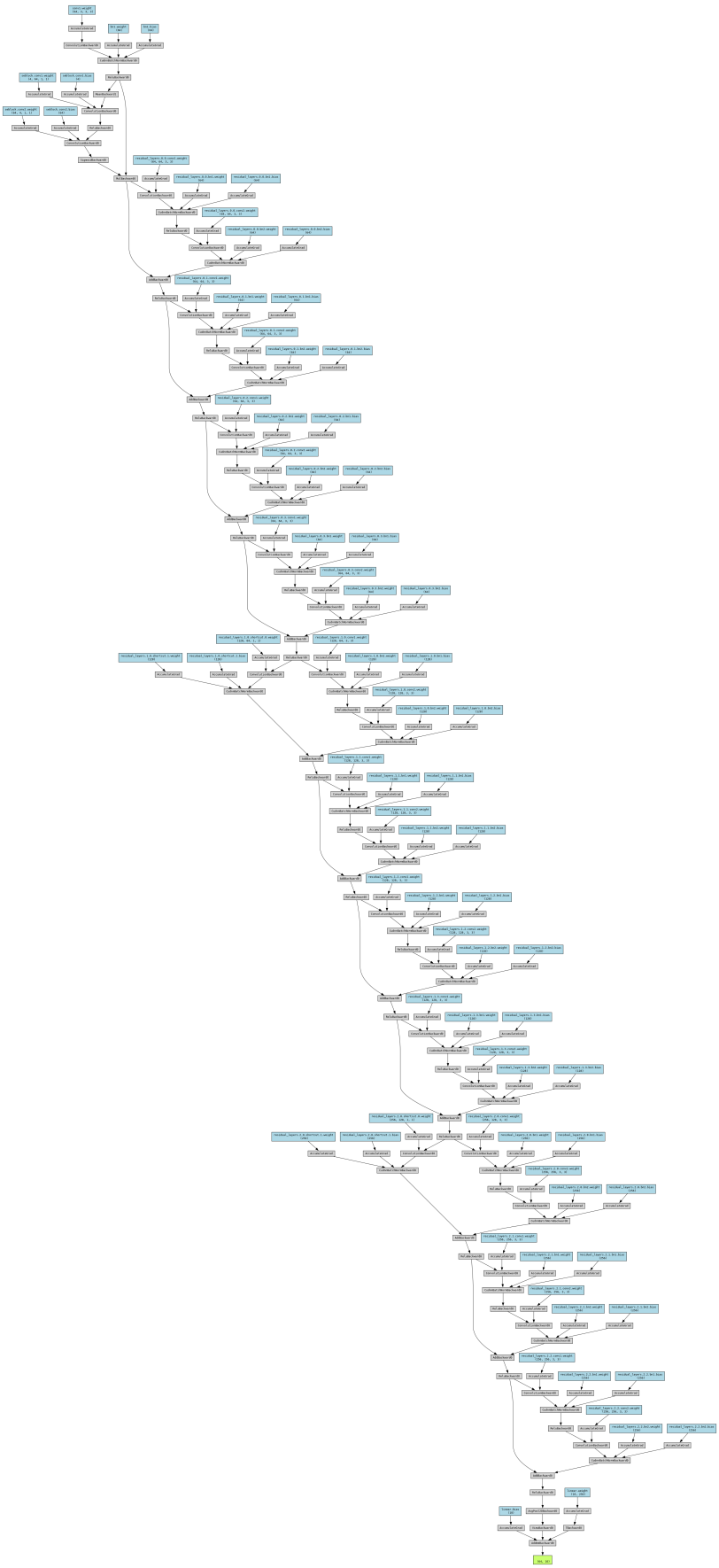


Figure 3: Visualization of the proposed architecure