Deep Learning Mini-Project

Harsh Chawla, Laksh Kataria, Ujjwal Vikram Kulkarni

New York University
Tandon School of Engineering
Brooklyn, New York 11201 USA
hc3338@nyu.edu, lvk8525@nyu.edu, uk2011@nyu.edu
Github Repository:https://github.com/harshchawla-26/DL_Mini_Proj

Abstract

The original ResNet18 architecture has over 11 Million parameters which requires a lot of computational power and resources. In this project, we compared several adaptations of ResNet18 architectures and performed parameter reduction techniques to reduce the parameters to under 5 Million while still attaining a competitive test accuracy.

Introduction

One of the popular beliefs in the domain of Deep Learning, and precisely image recognition, was "deeper the better". This statement implies that the deeper a neural network, the better it must perform at tasks such as image recognition and classification. Intuitively, this observation makes complete sense. For a deep neural network, the number of parameters in the said network would be exponentially high. Further, since neural networks are, in essence, used as function approximators, a higher number of parameters would facilitate better learning and consequently: better predictions.

This idea formed the basis for the VGG algorithm. VGG stands for Visual Geometry Group, the name of a group of researchers that came up with this architecture. Essentially, the VGG architecture stacks 2d convolution and max pooling layers in multiple blocks and tries to learn complex functions that would help it learn a particular image and classify similar images based on this understanding. However, the VGG architecture was plagued by a major problem. This problem was that of a "vanishing gradient". The title - "vanishing gradient", is pretty self-explanatory. Essentially, because of the deep architecture implemented in VGG, backpropagating the gradient through each of the layers, meant that by the time gradient updates were to be made for the initial layers, the value of the gradient was so small that hardly any learning was being performed, and training time was incredibly high!

This is why the ResNet architecture was introduced; to solve the vanishing gradient problem. ResNet stands for Residual Network, and the way it solves the vanishing gradient problem is by setting the local gradient for each gradient update to 1. The architecture does so by providing multiple skip connection channels between blocks of layers, which allow the gradient to flow unhindered, back to the initial layers. Not only does this greatly improve the quality of learning for a neural network and its generalizability over unseen data, but it also reduces training time, takes up fewer computational resources, and consumes lesser space, as compared to a general VGG architecture. Further, because of the skip connection architecture implemented in ResNet, the number of layers in the architecture could be increased to 50, 101, and 152 amongst other implementations.

The basic ResNet architecture can be seen in this image:

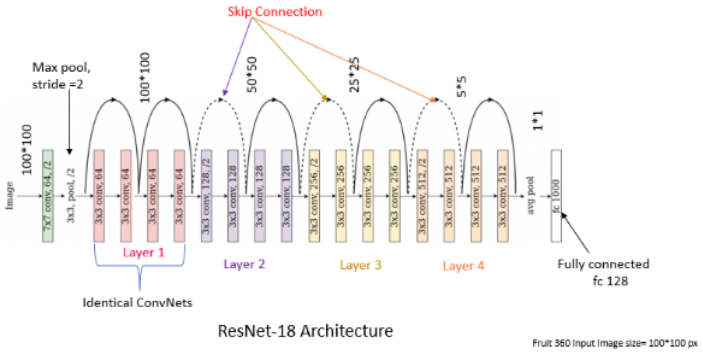

Figure 1: ResNet18 Architecture

As we can see, the input image is put through a convolution filter at first, followed by pooling and batch normalization operations, before it enters into the ResNet layer architecture. However, there is an interesting observation to be made here. The architecture shows just 4 'Layers', but each of these layers performs multiple operations. This is the idea behind 'blocks' in ResNet. Each block in a ResNet performs multiple operations, and each layer is composed of several blocks.

ResNet, however, comes with its issues. A generic ResNet-18 architecture has around 11.9 million parameters to train. Training this neural network requires high computational resources and takes quite a long time (relatively low compared to VGG, but still quite a while!). Most

GPU-based environments available online, such as Google Colab or Amazon's SageMaker, cannot handle such high computational requirements and fail to train a ResNet-18 architecture from scratch. For this reason, through this project, we aim to curtail the number of parameters in our ResNet-18 architecture, down to under 5 million. We do this by altering the operations within the channels, changing kernel sizes and stride lengths, and reducing the output dimensions at each layer. Through our optimization efforts, we finally arrive at an architecture that can be trained from scratch using online GPU environments and provides a competitive accuracy on our testing data.

Overview

In this project, we compared several adaptations of ResNet18 architectures. The first architecture had a base layer along with 4 layers of 2 blocks each, followed by a linear layer. For the convolutional layers, the number of channels is set as follows 3→64→64→128→256→242. This architecture consisted of 4.98M trainable parameters. After training for 200 epochs with a batch size of 32 we achieved a test accuracy of 86.43%. The second architecture had a base layer along with 4 layers of 2 blocks each, followed by the last layer being a linear layer as well. The number of channels for the convolution layers was the same as the first architecture. The kernel size and stride were different for this architecture. This architecture consisted of 4.99M parameters. This architecture was able to achieve 88.20% test accuracy after training for 100 epochs and a batch size of 16. The third and best performing model had 4 layers of 2 blocks each, with a dropout after each convolution, followed by a linear layer and a softmax activation. For the convolutional layers, the number of channels is set as follows 64→64→128→244→256. This architecture has 4.99M trainable parameters. This architecture achieved a test accuracy of 90.80% after training for 350 epochs with a batch size of 32.

Methodology

For this project, we work on the Cifar10 dataset. The Cifar10 dataset contains 60000 images of size 32x32. Each of these images belongs to one out of 10 classes. The data set has been divided into a training and a testing set. The training set contains exactly 50000 images, and the test set contains 10000. The test batch contains exactly 1000 images from each class and the training batches contain 5000 images from each class. We load this dataset using PyTorch for our project. Once the data has been loaded, we perform normalization operations on the images. We normalize these images by finding the means and standard deviations of the pixel values and replacing the original pixel values with values chosen from the resulting distribution. Normalization is an essential operation in an image classification task as it ensures that each input parameter, which in this case is a pixel value, has a similar distribution. This makes convergence while training the model much faster. Furthermore, we also perform some data augmentation techniques on the training data. Data augmentation is performed by adding random rotations to the images, random horizontal flips, and random crops. Augmentation adds some variation to the input images and prevents the network from overfitting to the training data. To perform these transformations, we use the transforms model from torchvision.transforms.

Once the data has been prepared, we then split the data into train, test, and validation data loaders. The items (images) from these loaders are then used to train and evaluate the model respectively.

As alluded to earlier, the model which we will be using for this purpose is the ResNet-18 architecture. We implement and subsequently train the model, from scratch. In our project, we tried three separate implementations; two with the same architecture and varying parameters, and one with a slightly modified architecture with an additional layer added for performing the softmax operation, dropout value set to 0.1, and varying output channel numbers for each of the 4 layers.

Method 1:

The first architecture [2] which we used, consisted of a 'Layer 0' which was used to perform convolution, pooling, and batch normalization operations, followed by 4 layers of 2 blocks each. These 4 layers were followed by a final linear layer which was designed such that the input to this layer, would be mapped to 10 output channels. The number of channels set for the initial layers and blocks were as follows, 3→64→64→128→256→242. For the final linear layer, the 242 output channels as mentioned earlier, were mapped to 10. For the first run of this architecture, we used a batch size of 256 images. The time taken per training epoch was somewhere around 33s which was highly desirable as we were training this model for 200 epochs. However, it was clear to see within the first few iterations that the validation accuracy did not follow the same curve as the training accuracy. Hence, we performed a re-run with batch size reduced to 128. Higher batch sizes reduce run time but could lead to the model overfitting to the train data, and this was the issue we were encountering. After reducing the batch size, we then changed the convolution kernel size for Layer 0 = 3 and stride = 1. These settings would ensure that the kernel was significantly small and covered more image pixels. Padding for the convolution was set to 1, however, this implementation was running into memory issues. To tackle this, we increased the size of the MaxPooling kernel to 5 and set stride = 3. A size 5 kernel with stride set to 3 is not ideal for learning, but it was a sacrifice we felt made sense as we allowed the convolution filter to capture as much information as possible and possibly make up for the information lost while pooling. We ran these settings for 200 epochs, achieving a training accuracy of 99.63%, but the validation hovered around $86 - 87\%$ indicating overfitting. The model had 4,984,152 and achieved a test accuracy of 86.44%

Method 2:

Another approach was the same architecture [2], with a base layer along with 4 layers of 2 blocks each, followed by the last layer being a linear layer. For the convolutional layers, the number of channels is set as follows 3→64→64→128→256→242 For the linear layers, the feature size is 242→10, with 10 being the number of classes. To improve accuracy from the previous architecture we tuned the kernel size, stride and padding of the base layer. For the convolution we set the stride to 7, stride to 1 and padding to 1 so that we could retain more information while training. Similarly, in the base pool layer, the kernel was reduced from 5 to 3 and the stride from 3 to 2, for the same reason to retain as much as information possible to train the model. This also increased the number of trainable parameters to 4,991,832. This increase in trainable parameters and information, led to CUDA memory running out while testing the model. To combat the CUDA error, we reduced the batch size from 32 to 16. The reduction in batch size improved the accuracy to 87% with Adam optimizer after training for 100 epochs. However, it increased the time for each epoch to approximately 1 min 22 seconds. We improved this model by using SGD with learning rate 0.01 and momentum as 0.9 as an optimizer and increased the test accuracy to 88.20% after 100 training epochs and each epoch taking about 1 min 6 seconds.

Method 3:

The Base ResNet18 Architecture was inspired from the repository mentioned in the description of the project. After running the proposed architecture, we found out that the memory required to run the code was exceeding the colab free tier limits. After multiple attempts we came to a realization that changing the stride value from 1 to 2 in the first convolution layer solved this problem.

Two design changes were made to the architecture [1]: A dropout layer was added in the residual block, and the value for the dropout parameter was set to 0.1. The dropout parameter can be used per layer in a dense neural network architecture and helps solve the problem of overfitting. Setting the value of the dropout to 1.0 means there would be no dropout, and having a value closer to 0.5 gives an equal probability to retain the values from the nodes. Next, we added a softmax layer at the end to compute the output.

The model had the following output channels for the convolutional layers 3→64→64→128→244→256, and the kernel and stride values were set to 3 and 2 for the initial base convolution layer. The optimizer used for running this architecture was SGD with a learning rate of 0.01 and momentum = 0.9. The architecture was trained for 350 epochs, with batch size = 32, and the training took over 4 hours to complete. The training accuracy observed for this model was 99.5% and we ended up with a test accuracy of 90.80%. A summary of the various combinations of parameters we used for this architecture has been shown below in Table 1.

Results

The results of our final model have been summarised in this section. To present a clear picture of our results and our inference from the same, we have included a few graphs and plots.

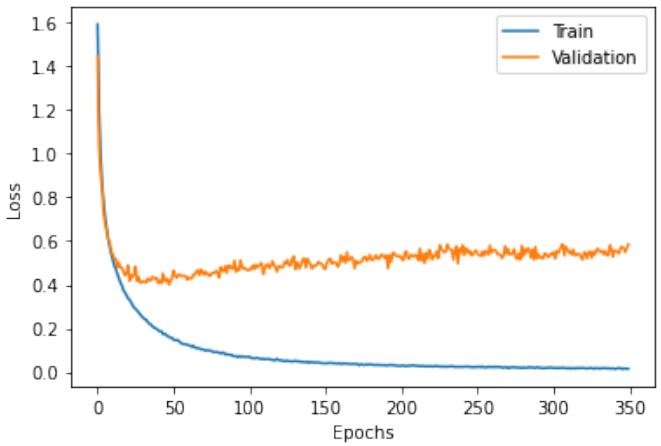

Figure 2: Training Loss vs Validation Loss

Figure 2 represents the training loss vs the validation loss, as observed while training the ResNet architecture, and Figure 3 respresents the Training and Validation accuracies. As mentioned earlier, the ResNet architecture was introduced to solve the issues related to vanishing gradient, observed in denser implementations such as VGG, however, from the plots of loss and accuracy, we can see that the validation loss and accuracy plateaus at a value higher and lower than the corresponding training values. The validation loss in fact, shows a slight increasing trend with an increase in the number of epochs.

The best validation accuracy is observed around 90%

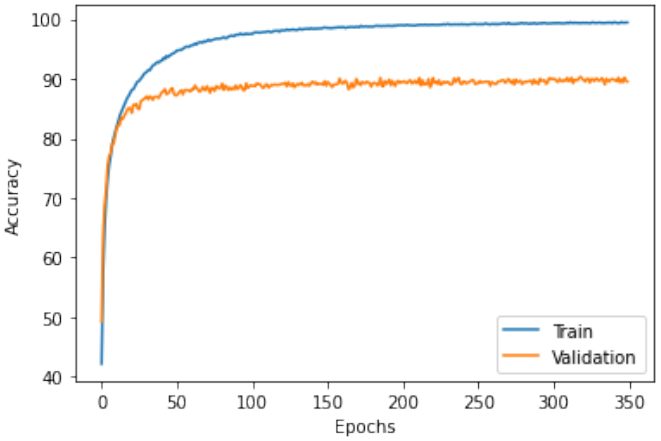

Figure 3: Training Accuracy vs Validation Accuracy

Results					
Batch Size	Epochs	1 Epoch Time	Learning Rate	Training Accuracy	Test Accuracy
256	300	33	0.01	99.3	89.56
128	300	38	0.01	99.3	89.70
64	350	42	0.005	99.3	89.9
32	350	46	0.005	99.4	90.39
32	350	46	0.01	99.5	90.80

Table 1

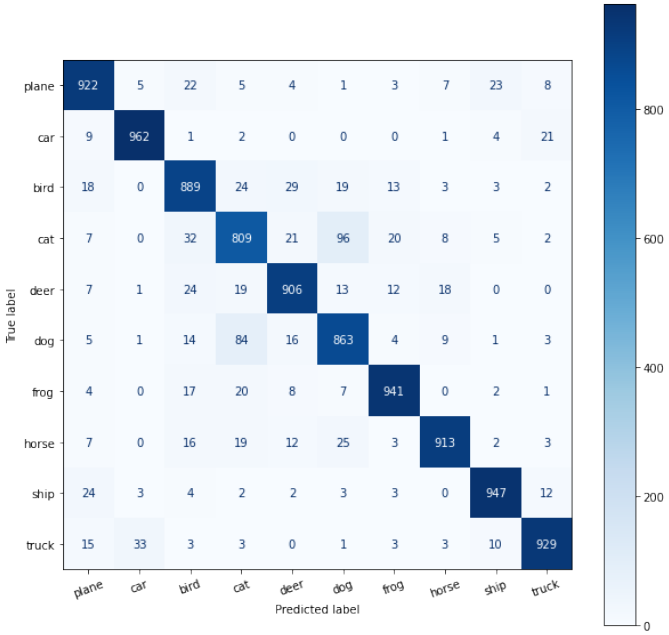

Figure 4: Confusion Matrix

Finally, we have included the confusion matrix plot for the classification algorithm. Our model returns an accuracy of 90.80% on the test data and the confusion matrix corroborates this observation.

References

[1] https://github.com/kuangliu/pytorch-cifar

[2] https://github.com/liao2000/ML-Notebook/blob/main/ResNet/ResNet_PyTorch.ipynb