CSCE 636: Deep Learning Project

Akhilesh Vijaykumar
Department of Visualization
Texas A&M University
College Station, Texas
`akhivijay97@tamu.edu`

Abstract

The goal of this project is to build a novel learner to classify the CIFAR-10 dataset. The primary focus is the application of optimization strategies to a Residual Neural Network (ResNet).

1 Dataset

The CIFAR-10 dataset [1] consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order. Between them, the training batches contain exactly 5000 images from each class. The training images alone have been used to tune the novel learner.

Images from each class in the dataset are shown in Figure 1.

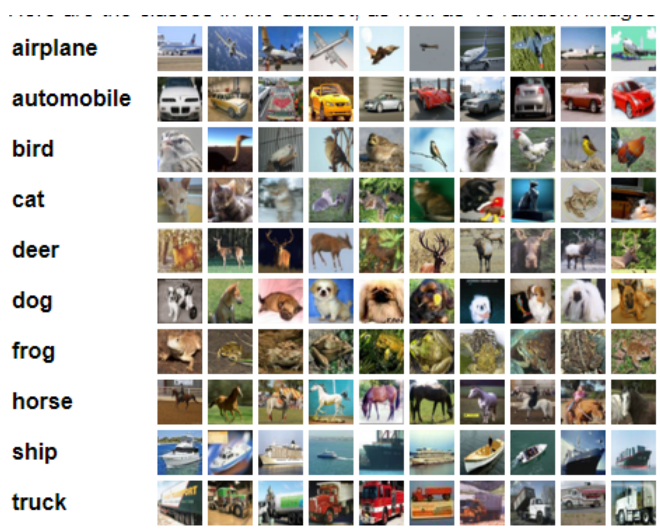

Figure 1: CIFAR-10 Dataset

2 Software Requirements

The high-level object-oriented programming language Python is used in this project. The open source software heavily utilized throughout this project is Tensorflow 2.2. The prior package comes standard in Python distributions and can be installed similarly if necessary.

3 Project Structure

This section discusses the organization and code structure of the project.

3.1 Organization of Project

The *code* directory contains all of the files required to produce the final predictions seen in this report.
The *data* sub-directory contains the training data used to tune the novel learner, the public test data and the private test data.
The *saved_models* sub-directory contains the weights of final model.
The *predictions.npy* file contains the predictions on the private test data provided.

3.2 Structure of Source Code

The *code* directory contains the following files:

main.py: Includes the code that loads the dataset and performs the training, testing and prediction.
DataLoader.py: Includes the code that defines functions related to data I/O.
ImageUtils.py: Includes the code that defines functions for some pre-processing of the images.
Configure.py: Includes dictionary that set the model configurations. The dictionary is imported to *main.py*.
Model.py: Includes the code that defines the model in a class. The class is initialized with the configuration dictionaries and should have the methods "train", "evaluate", "predict_prob". The defined model class is imported to and referenced in *main.py*.
Network.py: Includes the code that defines the ResNet architecture. The defined network is imported and referenced in *Model.py*. The code for building the ResNet has been sourced from [2].

4 Novel Learner

4.1 Literature Review

The authors of [3] explored the effects of various data augmentation methods on image classification when using a Deep Convolution Neural Network. The Alexnet model was used as the network model and a subset of CIFAR10 and ImageNet (10 categories) were selected as the data set. The data augmentation methods explored were : Generative Adversarial Network (GAN/WGAN), Flipping, Cropping, Shifting, PCA jittering,Color jittering, Noise, Rotation, and some pairwise and triplet combinations. Experimental results showed that, Horizontal Flipping and Random Cropping was the pair that produced the most improvement in accuracy.

The authors of [4] explored learning rate heuristics of (cosine) restarts and warmup. Their analysis suggests showed that: the success of cosine annealing are not often seen in practice, the effect of learning rate warmup is to prevent the deeper layers from creating training instability.

The cosine decay strategy decays learning rates along a cosine curve and then, at the end of the decay, restarts them to its initial value. The learning rate at the t-th epoch is given by:
$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + \cos(\frac{T_{curr}}{T_i}\pi))$
where η_{min} and η_{max} are the lower and upper bounds respectively for the learning rate. T_{curr} represents how many epochs have been performed since the last restart and a warm restart is simulated once T_i epochs are performed. Also $T_i = T_{mult} \times T_{i-1}$, meaning the period T_i for the learning rate variation is increased by a factor of T_{mult} after each restart.

While the strategy has been claimed to outperform other learning rate schedulers, little is known why this has been the case. One explanation that has been given is that a restart will help get out of a local optimum and explore another region.

The authors of [5] proposed *mixup*, a simple learning principle to eliminate undesirable behaviors such as memorization and sensitivity to adversarial examples.Mixup trains a neural network on convex combinations of pairs of examples and their labels. By doing so, mixup regularizes the neural network to favor simple linear behavior in-between training examples. Their experiments on the ImageNet-2012, CIFAR-10 , CIFAR-100, Google commands and UCI datasets show that mixup improves the generalization of state-of-the-art neural network architectures. They also found that *mixup* reduces the memorization of corrupt labels, increases the robustness to adversarial examples, and stabilizes the training of GANs.

mixup constructs virtual training examples:
$\hat{x} = \lambda x_i + (1 - \lambda)x_j$, where x_i, x_j are raw input vectors
$\hat{y} = \lambda y_i + (1 - \lambda)y_j$, where y_i, y_j are one-hot label encodings.
(x_i, y_i) and (x_j, y_j) are two examples drawn at random from training data, and $\lambda \in [0, 1]$.

Therefore, *mixup* extends the training distribution by incorporating the fact that linear interpolations of feature vectors should lead to linear interpolations of the associated targets.also, *mixup* introduces minimal computation overhead and despite its simplicity, allows a new state-of-the-art performance , increases the robustness of neural networks when learning from corrupt labels, or facing adversarial examples, improves generalization on speech and tabular data, and, can be used to stabilize the training of GANs.

The authors of [6] show that the key to adopt a deep model to a small dataset, is to use Batch Normalization and strong Dropout setting. For fast convergence without very high accuracy, Batch Normalization on a very deep model can be used and the model will gain comparable accuracy with shallow models and converges faster. For high accuracy, both Batch Normalization and strong Dropout setting can be used on a very deep model, to gain high accuracy with relatively fast convergence.

4.2 Architecture

A depiction of the novel learner design is shown in Figure 2.
The learner is a ResNet: version 1, that uses the original residual blocks from Figure 4(a) in [7] .
It has a depth of 3, with 3 Residual blocks per stack.

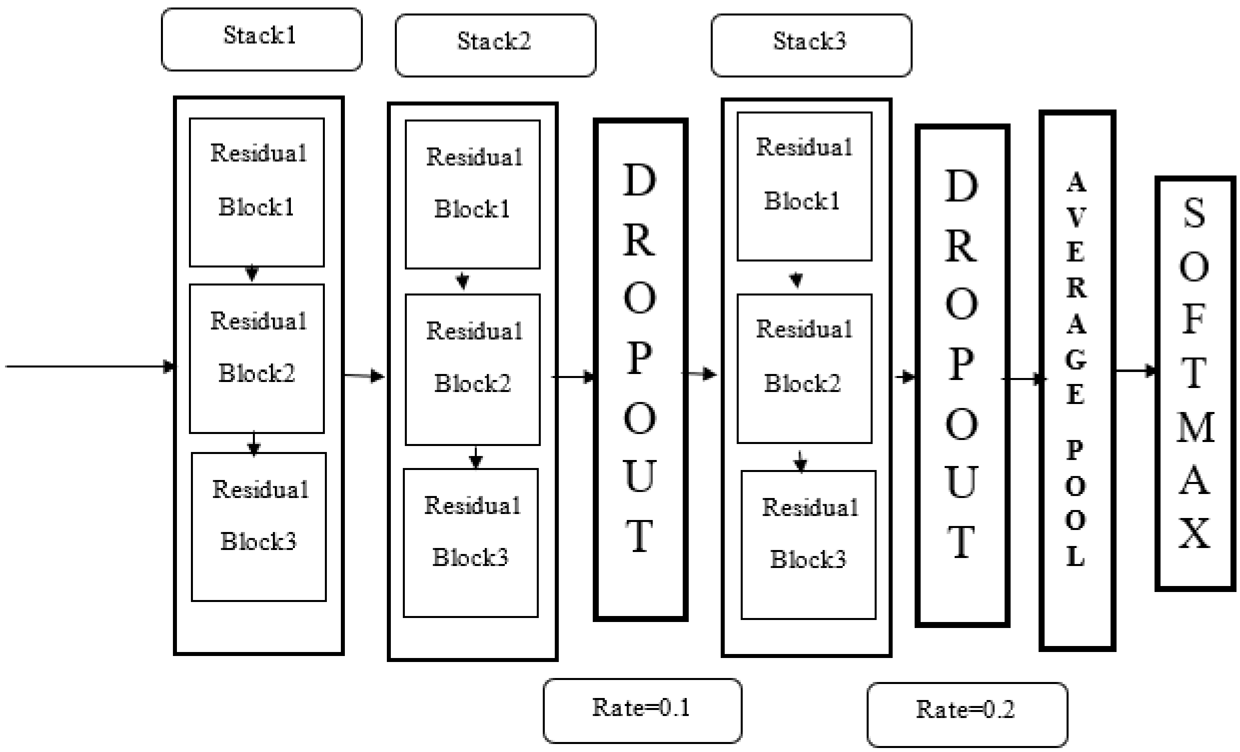

Figure 2: Novel Learner Architecture

4.3 Optimizations

This section outlines the optimizations employed to find the best hyper-parameters and achieve the best accuracy on the novel learner. The base configuration is a simple ResNet with a depth of 3 and 3 Residual blocks per stack, and is used as the standard to compare with. The model learns on the training data provided in batches, each of size 32 examples, and the impact of the optimizations is evaluated for 200 epochs on the public test data.

The first optimization performed was the use of *mixup* training. The accuracies of the learner on the public test data before and after *mixup* training shown in Table 1.

Table 1: Accuracies before and after *mixup* training

Mixup	Accuracy (%)
With	**89.54**
Without	88.21

The above results verify that *mixup* training helps the model generalize better on training data.

The second optimization performed was the use of cosine learning rate decay.
The accuracies of the learner on the public test data before and after cosine decay is shown in Table 2.

Table 2: Accuracies before and after cosine decay

Cosine decay	Accuracy (%)
With	**89.84**
Without	89.54

The third optimization performed was the use of Dropout layers between stacks of residual blocks. 10% of the connections are randomly dropped after Stack2 and 20% of the connections are randomly dropped after Stack3.

The fourth optimization performed was experiments with different Optimizers. The accuracies of the learner on the public test data for each optimizer *mixup* is shown in Table 3.

Table 3: Accuracies for different optimizers

Optimizer	Accuracy (%)
Adam	**90.12**
RMSProp	89.84
SGD	66.4

4.4 Novel Model Results

The novel learner achieves an accuracy of **90.12%** on the public test data.

5 Conclusion and Future work

The studies in this project focused on creating and optimizing a Residual Neural-Network. A possible way to extend this work in the future is to further optimize the model with more extensive hyper-parameter space searches and data augmentations.

References

[1] "Cifar10 dataset - https://www.cs.toronto.edu/ kriz/cifar.html,"

[2] "Trains a resnet on the cifar10 dataset - https://keras.io/zh/examples/cifar10_resnet/,"

[3] J. P. H. S. Jia Shijie, Wang Ping, "Research on data augmentation for image classification based on convolution neural networks," *IEEE*, 2018.

[4] C. X. R. S. Akhilesh Gotmare, Nitish Shirish Keskar, "A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation," 2018.

[5] Y. N. D. D. L.-P. Hongyi Zhang, Moustapha Cisse, "mixup: Beyond empirical risk minimization," 2018.

[6] W. D. Shuying Liu, "Very deep convolutional neural network based image classification using small training sample size," 2015.

[7] "Identity mappings in deep residual networks - https://arxiv.org/abs/1603.05027,"