Ablation Study on the Effect of Learning Rate, Learning Rate Scheduling, Weight Decay, and Data Augmentation with MobileNet and CIFAR-100

Abstract

In this study, we conducted an ablation analysis to investigate the impact of learning rate, learning rate scheduling, weight decay, and data augmentation on the performance of MobileNet architecture using the CIFAR-100 dataset. We hypothesized that these factors would significantly contribute to the model's accuracy and generalization ability. To evaluate this hypothesis, we designed a series of experiments where we systematically varied each factor while keeping the others constant. We trained multiple instances of MobileNet, each with a different combination of these factors, and measured their performance in terms of classification accuracy.

Introduction

Many good-performing deep learning architectures have recently existed for various computer vision tasks, including image classification. However, employing a sophisticated structure could result in challenges such as overfitting if proper parameter selection and regularization techniques are not specified. In this research, our focus is to examine the impact of learning rate scheduling, weight decay, and data augmentation on the performance of the MobileNet architecture when it is implemented.

Experiment Setup

In this experiment, we utilize MobileNet as the architecture and CIFAR-100 as the dataset. In addition, our chosen optimization algorithm is stochastic gradient descent with a momentum value of 0.9, which ensures fast convergence and helps to prevent getting stuck in local minima during training. Furthermore, during training of the network, a batch size of 128 is employed.

MobileNet

MobileNet is a lightweight convolution neural network model developed by researchers at Google in the paper "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications".[1]MobileNet used depthwise separable convolutions and linear bottlenecks to achieve a more petite model size and computational efficiency while maintaining competitive accuracy. The depthwise separable convolution reduces the computational cost by 1/N+1/K, where N is the number of kernels and K is the kernel size.

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Figure 1. MobileNet Architecture

Figure 1 shows the architecture of MobileNet, where the input passes through several depthwise separable convolutional layers, followed by a global average pooling layer and a fully connected layer. To ensure consistency and reproducibility in our experiments, we used a random seed value of zero for the entire experiment.

CIFAR-100

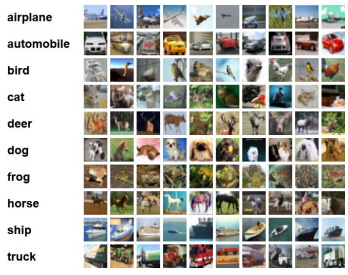


Figure 2. Example images from CIFAR-100

CIFAR-100 is a widely used Computer Vision benchmark dataset that consists of 100 classes, with each class containing 600 examples. Figure 2 shows some example images from the CIFAR-100 dataset.

Data Preprocessing

The CIFAR-100 dataset used in this experiment is obtained from torchvision, which is separated initially into a training set and a test set, with 50000 and 10000 samples, respectively. For this experiment, we will partition the dataset into three sets: training, validation, and testing, which consist of 40,000, 10,000, and 10,000 samples, respectively. So, we further split the original training set into the training and validation sets with a ratio of 80:20. The random seed of value 0 is used as mentioned in the Experiment Setup.

```
training_set_mean = (0.5071, 0.4865, 0.4409)
training_set_std = (0.2673, 0.2564, 0.2762)
2 usages
def get_train_valid_loader(dataset_dir, batch_size, mixup, seed, save_images):
    training_transformation = transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(training_set_mean, training_set_std)
    ])
    valid_transformation = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(training_set_mean, training_set_std)
    ])
    train_set = CIFAR100(root=dataset_dir, train=True, download=True)_# transform=

    train_set, validation_set = random_split(
        train_set, [40000, 10000], generator=torch.Generator().manual_seed(seed))
    train_set = MyDataset(train_set, transform=training_transformation)
    validation_set = MyDataset(validation_set, transform=valid_transformation)
    train_loader = DataLoader(train_set, batch_size=batch_size)
    valid_loader = DataLoader(validation_set, batch_size=batch_size)
    return train_loader, valid_loader

2 usages
def get_test_loader(dataset_dir, batch_size):
    transformation = transforms.Compose([
        # transforms.RandomCrop(32, padding=4),
        # transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(training_set_mean, training_set_std)
    ])
    test_set = CIFAR100(root=dataset_dir, train=False, download=True, transform=transformation)
    test_loader = DataLoader(test_set, batch_size=batch_size)
    return test_loader
```

Figure 3. Code for custom dataset loaders

For convenience, we created two custom dataset loaders called get_train_valid_loader and get_test_loader, as shown in Figure 3. This function implements the separation of the training set, data whitening, and augmentations, including random horizontal flip with a flipping probability of 0.5, random crop with dimensions of 32x32, and 4 pixels padding using torchvision.transforms module. Since the augmentation is to help the model generalize better, we only apply it to the training set and not the validation or testing sets. The training set is first split by the random_split function from torch.util.data module into training and validation sets. A subclass of dataset called MyDataset is created to hold the split set and apply the necessary transformations separately. Otherwise, the same transformation would be applied to both the training and validation sets, which would undermine the credibility of the validation process. Furthermore, data whitening requires a calculation of mean and standard deviation across the training set, but calculating these values every time during the experiment would be time-consuming. Therefore, we pre-calculate the mean and standard deviation of the training set in a separate script, "cifar-mean-std.py," and pass these values to the dataset loader. The obtained values are then used to perform data whitening on the input images during training and testing. We calculated the mean to be (0.5071, 0.4865, 0.4409) and the standard deviation as (0.2673, 0.2564, 0.2762).

The techniques mentioned above will be used throughout the project to preprocess the data and augment the training set.

Class	New_Portion	Class	New_Portion	Class	New_Portion
bed	1.00%	baby	0.99%	seal	0.98%
pine_tree	1.02%	pear	1.05%	girl	0.99%
clock	0.96%	cloud	0.99%	mouse	1.04%
spider	0.98%	raccoon	1.03%	turtle	0.99%
trout	1.01%	plain	1.04%	road	1.04%
shrew	1.02%	hamster	1.00%	crab	1.00%
dolphin	0.98%	rose	0.99%	sea	1.00%
orchid	0.98%	lamp	0.99%	tulip	0.99%
willow_tree	1.00%	lizard	1.02%	train	0.97%
aquarium_fish	0.99%	chimpanzee	1.08%	table	1.03%
maple_tree	0.98%	wolf	0.96%	sweet_pepper	1.00%
oak_tree	0.97%	bicycle	1.01%	snail	1.01%
mountain	0.97%	lion	1.00%	man	1.01%
palm_tree	1.00%	leopard	0.96%	plate	1.02%
sunflower	1.03%	tiger	0.94%	television	1.02%
woman	0.95%	camel	1.05%	cockroach	0.97%
tank	1.01%	bus	1.01%	worm	1.00%
snake	0.97%	otter	1.01%	bear	0.99%
house	1.04%	butterfly	1.01%	bottle	0.98%
fox	1.00%	squirrel	1.03%	rocket	1.00%
porcupine	1.01%	lawn_mower	1.03%	lobster	1.02%
apple	0.94%	poppy	1.01%	crocodile	1.02%
cup	0.99%	bowl	1.01%	orange	1.00%
couch	1.02%	beetle	0.98%	castle	1.06%
rabbit	0.99%	whale	1.01%	flatfish	0.99%
bridge	0.99%	can	0.98%	dinosaur	0.98%
cattle	1.03%	shark	0.97%	motorcycle	1.05%
bee	1.00%	kangaroo	1.03%	keyboard	1.00%
boy	0.99%	possum	1.03%	ray	1.02%
telephone	0.97%	beaver	1.01%	tractor	0.98%
caterpillar	1.00%	skunk	0.98%	mushroom	1.01%
wardrobe	1.01%	pickup_truck	0.97%	skyscraper	1.01%
streetcar	1.03%	elephant	0.97%	forest	0.97%
				chair	0.97%

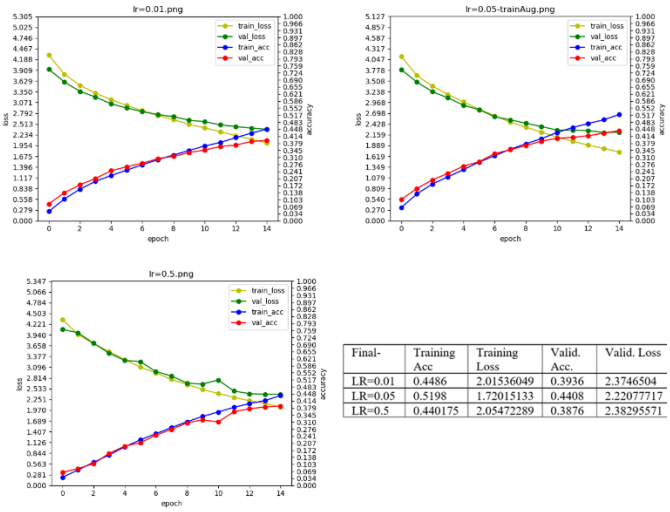
Figure 4. New class proportion of the training set

Figure 4 shows the new class proportion after splitting the training set into training and validation sets. The visualization is performed in a file called “class_portion.py”.

Tuning Learning Rate

The learning rate is a crucial hyperparameter, as it determines the speed of the gradient descending towards the global minimum of the loss function. It can be represented by $\theta = \theta - \alpha * \nabla L(\theta, xi)$, where α denotes the learning rate. It is important to find an optimal value for the learning rate in

order to achieve fast and effective convergence of the neural network. The choice of a suitable learning rate depends on the architecture of the model, the characteristics of the dataset, and the optimization algorithm, which makes it a challenging task, and no standard approach exists. The best way to tune the learning rate is through experimentation and iterative refinement. In this experiment, three initial learning rates (0.5, 0.05, 0.01), are selected as candidates for the learning rate. To find out the best learning rate, MobileNet was trained on the CIFAR100 dataset using each of these initial learning rates with 15 epochs.



Final-	Training Acc	Training Loss	Valid. Acc.	Valid. Loss
LR=0.01	0.4486	2.01536049	0.3936	2.3746504
LR=0.05	0.5198	1.72015133	0.4408	2.22077717
LR=0.5	0.440175	2.05472289	0.3876	2.38295571

Figure 5. Training and validation accuracy and loss of learning rate equal to 0.01 (top left), 0.05 (top right), and 0.5 (bottom left), Table 1. final accuracy and loss (bottom right)

In Figure 5, the graph displays the accuracy and loss of the training and validation sets for three different initial learning rates R: 0.01 (top left), 0.05 (top right), and 0.5 (bottom left). The final loss and accuracy for both the train and validation sets are listed in the table (bottom right). The final training and validation loss for learning rates of 0.01 and 0.5 are similar, at approximately 2.0 (train) and 2.37 (validation). Similarly, the final accuracies for both learning rates are around 0.44 (train) and 0.39 (validation). Besides, the final losses for LR=0.05 are 1.7 (train) and 2.22 (validation), with final accuracies of 0.52 (train) and 0.44 (validation). Obviously, the learning rate of 0.05 yielded the best results in terms of both final loss and accuracy on the train and validation sets.

Pattern-wise, the training and validation loss of LR=0.01 have a gradually decreasing curve and look quite stable. However, compared to LR=0.05, the training and validation loss of LR=0.05 reduced slightly more rapidly while remaining relatively stable throughout the training process. The slower convergence of the learning rate of 0.01 is due to the smaller step size it takes during weight updates, resulting in slower learning. And could potentially get stuck in a suboptimal solution. On the other hand, when the learning rate is set to 0.5, the training and validation loss fluctuate, indicating that the chosen initial learning rate is too large that it might possibly jump between different local minima and struggle to converge.

Therefore, the learning rate of 0.05 was found to be the best choice in terms of achieving both low loss and high accuracy on the train and validation sets while balancing the convergence speed and stability of the training process.

Learning Rate Scheduling

Learning rate scheduling is a technique used to adjust the learning rate during the training process to avoid fluctuating convergence and lead to faster and more stable convergence.

$$LR = LR_{\min} + 0.5(LR_{\max} - LR_{\min})(1 + \cos((\pi T_{current})/T)$$

(Equation 1.)

In this experiment, cosine annealing learning rate scheduling was used, where the learning rate decreases in a cosine-like pattern as training progresses. Cosine annealing gradually reduces the learning rate from LR_{max} to LR_{min} over the span of N epochs. In our case, LR_{max} =0.05 and LR_{min} =0. The formula for cosine annealing is denoted by (Equation 1.). Where:

- LR_{max} and LR_{min} represent the lower and upper bounds of the learning rate, respectively.
- $T_{current}$ represents the current epoch.
- T denotes the total number of epochs.

We conducted two sets of experiments using the MobileNet model to analyze the impact of cosine annealing. The "Experiment Setup" was utilized for both trials, with a learning rate of 0.05 based on previous ablation study findings. Each trial consisted of 300 epochs, one with cosine annealing and another without it.

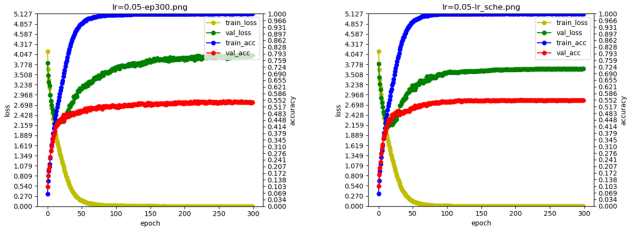


Figure 6. Training-validation accuracy and loss for 300 epochs, (left) without cosine annealing, (right) with cosine annealing

	Train loss	Valid. Loss	Train acc.	Valid acc.
w/o cos	1.69e-04	4.0577	0.99995	0.5477
Cos	2.71e-04	3.65746	0.999975	0.5492

(Table 2. Final train-validation loss and accuracy with and without cosine annealing)

Figure 6 depicts the training-validation accuracy and loss for 300 epochs, and Table 2 shows the final train-validation loss and accuracy, comparing the results with and without cosine annealing. It is observed from Table 2, though not obvious, that the model trained with cos annealing has slightly higher accuracy and lower loss. However, the significant difference between the accuracy of the training and validation datasets, along with the lack of substantial improvement in accuracy, indicates that implementing cosine annealing did not effectively address overfitting within the model.

Moreover, when utilizing cosine annealing for learning rate scheduling, Figure 6 demonstrates that the validation accuracy and loss curves exhibit smoother patterns compared to cases without such scheduling. There is a noticeable fluctuation in the validation accuracy and loss when learning rate scheduling is not applied. This is likely due to the fact that cosine annealing allows the model to navigate closer to the optimal solution with smaller steps when usually close to minima towards the end of training. Eventually, cosine annealing is effective in improving the model's stability for convergence to minima.

Weight Decay

As mentioned earlier, MobileNet has an overfitting problem due to its complex architecture, while weight scheduling did not significantly address this issue. Regularization techniques such as weight decay can be effective in mitigating overfitting. Often, weight decay and L2 regularization easily cause confusion. The difference between them is that weight decay directly adds the penalty term to the weight update rule, while L2 regularization adds it to the loss function; weight is commonly used due to its computational efficiency. Therefore, weight decay is opted for this experiment.

To investigate the impact of weight decay on the MobileNet model, another set of experiments was conducted with two weight decay coefficients; they are lambda = 0.0001 and 0.0005. All other configurations remain consistent with the previous section, including incorporating a learning weight schedule.

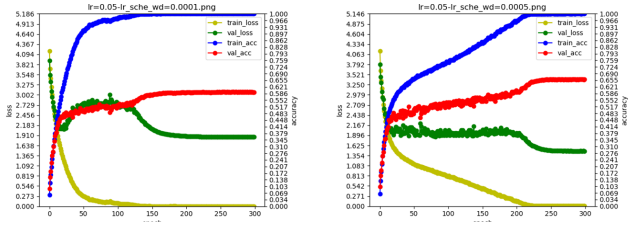


Figure 7. Training-validation accuracy and loss curves with weight decay 0.0001 (left) and 0.0005 (right)

	Train loss	Valid. Loss	Train acc.	Valid acc.
0.0001	0.0025	1.8655	1.0000	0.5936
0.0005	0.0091	1.4757	1.0000..	0.6592

(Table 3. Final train-validation loss and accuracy with weight decay 0.0001 (left) and 0.0005 (right))

From Table 3, the final loss and accuracy demonstrate an exceptional improvement when weight decay is applied compared to the result with cosine annealing only. The validation loss was greatly reduced from 3.6 to 1.8 (lambda=0.0001) and 1.4 (lambda=0.0005), and accuracy has significantly increased from 0.54 to 0.59 (lambda=0.0001) and 0.65 (lambda=0.0005). Also, it is obvious that 0.0005 is a better weight decay coefficient as it has a drastic improvement in the sense of accuracy and loss.

To investigate the reason for the improvement observed with weight decay, it is important to understand the root cause of overfitting. Overfitting occurs when a model is too complex and starts to memorize the training data rather than learning generalizable patterns. Weight decay penalizes the update when the magnitude of the weights becomes too large, shrinking them towards zero, which simplifies the function represented by the model. The loss and accuracy curve in Figure 7 depicts the effect of the penalty. The loss curve in the Figure 6 section begins to curl upward after around 50 epochs, indicating an overfit; after implementing weight decay, the curling effect of the loss curve is suppressed with lambda=0.0001 and ceased with a heavier weight decay coefficient of 0.0005, indicating that the model generalizes better. The training loss and accuracy also depict the regularization power of lambda=0.0005, that both curves converge after around 200 epochs, which the model converges after 50 epochs without weight decay.

Mixup Augmentation

Although weight decay has shown significant improvements in preventing overfitting, there still exists overfitting in the model, as seen from the discrepancy, around 35%, between the training and validation accuracy in Figure 6. To further address this issue, mixup data augmentation is applied.

$$x = \lambda x_i + (1-\lambda)x_j$$
$$y = \lambda y_i + (1-\lambda)y_j$$

(equation 2. mixup interpolation)

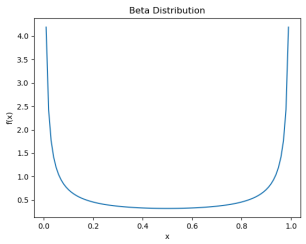


Figure 8. Beta pdf

Figure 9. An example of a mixup image: a plane and a car

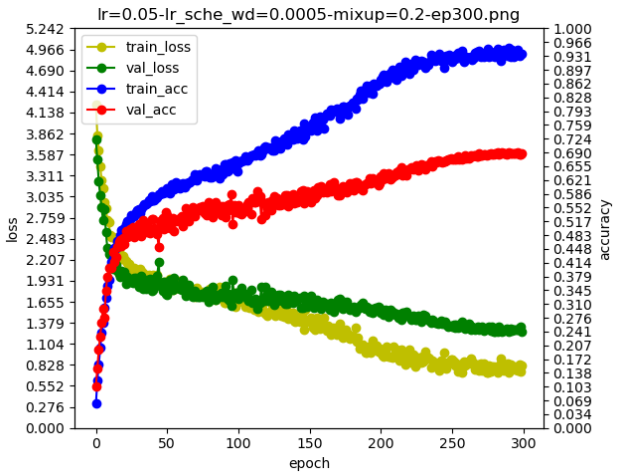


Figure 8. Training-validation accuracy and loss curves with Mixup

Mixup data augmentation generates virtual training samples by linearly interpolating between pairs of training samples. The idea is rather simple: both input vectors and labels undergo (2) to produce new synthetic samples. It's worth mentioning the labels are required to be converted to one hot encoding; otherwise, interpolating them would be meaningful and generate wrong labels. The interpolation ratio of each pair follows a beta distribution parameterized by α and λ, where α controls the shape of the distribution and λ is sampled from the beta distribution. In this experiment, the beta is set to 0.2. The probability density function of beta=0.2 is shown in Figure 8. It shows that, most likely, a class will dominate the ratio. An example of a mixup image (plane and car) is illustrated in Figure 9, with a lambda of 0.25.

	Train loss	Valid. Loss	Train acc.	Valid acc.
mixup	1.2897	1.3442	0.8869	0.6875

(Table 4. Final train-validation loss and accuracy with Mixup)

According to Table 4, the validation loss and accuracy obtained are 1.26706128 and 0.6881, which have slightly improved compared to weight decay. Moreover, the loss and accuracy are 0.82028925 and 0.93715, which are not 0 and 1 anymore. Furthermore, As shown in Figure 10, the curves of train-validation loss and accuracy exhibit gradual slopes and a more consistent learning pattern. These are evidences that mixup augmentation has expanded the regularization power of the model.

The root causes of its effectiveness are two-fold. First, Mixup helps reduce the need for memorizing incorrect labels, which can harm the learning process. Secondly, it enhances the model's ability to handle adversarial examples, which are inputs specifically designed to mislead the model. This increases the model's robustness for more general data.

Conclusion

In conclusion, this study conducted an ablation analysis to examine the impact of learning rate, learning rate scheduling, weight decay, and data augmentation on the performance of the MobileNet architecture using the CIFAR-100 dataset. The results indicate that these factors significantly contribute to the model's accuracy and generalization ability. Specifically, tuning the learning rate and scheduling it can improve the model's performance considerably. Using weight decay and data augmentation can also help prevent overfitting and improve the model's generalization ability. Overall, this study provides valuable insights into the factors that can affect the performance of the MobileNet architecture and can guide future research in this area.

References

[1] "[1704.04861] MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", arxiv.org, (Accessed 24 Oct. 2023).