AI6103 - Homework Assignment

Bugnot Reinelle Jan Cruz

School of Computer Science and Engineering
G2304329L
bugn0001@e.ntu.edu.sg

Abstract

This paper aims to investigate the effects of hyperparameters such as initial learning rate, learning rate schedule, weight decay, and data augmentation on MobileNet, trained on the CIFAR-100 dataset.

MobileNet

The Deep Learning model that this report will use across all experiments is *MobileNet*, a convolutional neural network (CNN) tailored for mobile and embedded vision applications (Pujara 2023). It was open-sourced by Google and is TensorFlow's first mobile computer vision model. The core architecture of MobileNet is based on a streamlined architecture that uses depthwise separable convolutions which significantly reduces the number of parameters and computation time compared to other networks, resulting in a lightweight deep neural network (Howard 2017).

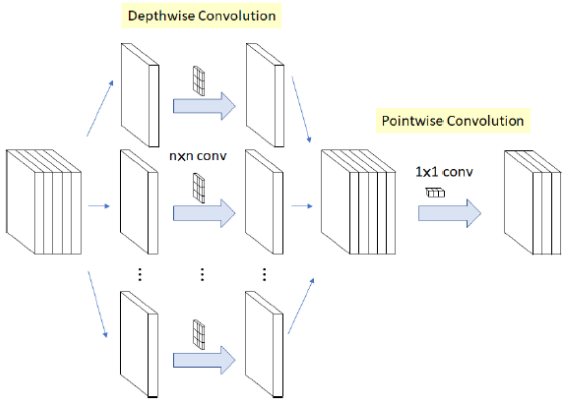

Figure 1: MobileNet: Depthwise + Pointwise Convolution

As shown in Fig. 1, depthwise separable convolution, the key component of MobileNet, is made from two operations: depthwise convolution and pointwise convolution. This convolution originated from the idea that a filter's depth and spatial dimension can be separated, hence, the name separable. The depthwise convolution applies a single filter to each input channel, and the pointwise convolution then applies a 1x1 convolution to combine the outputs of the depthwise convolution (Howard 2017).

The motivation behind MobileNet's development was to strike a balance between the accuracy of complex neural networks and the performance limitations of mobile runtimes. Given the restricted computational resources of mobile and embedded devices, mobile deep learning models must be efficient. MobileNet addresses this need by significantly reducing parameter count and computation time, making it well-suited for training classifiers that prioritize both size and speed. Consequently, it is particularly advantageous for mobile applications where minimizing model size and latency are critical factors (Pujara 2023).

CIFAR-100

The MobileNet model will be trained on the CIFAR-100 dataset, which consists of 60,000 color images, each measuring 32x32 pixels, and is categorized into 100 distinct classes. With 600 images per class, it maintains a balanced distribution across its classes. These 600 images are further split into training and testing sets, with 500 training images and 100 testing images per class. Despite its small size, the dataset presents realistic images, rendering it a widely utilized and challenging dataset for various machine learning and computer vision applications.

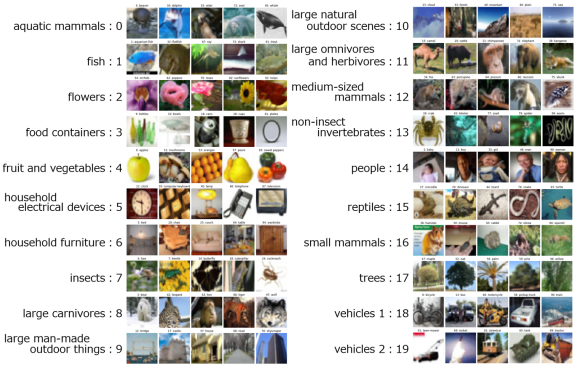

Figure 2: CIFAR-100 and its 20 course labels. Each course label is further divided into 5 fine labels, producing a total of 100 labels (Krizhevsky and Hinton 2009)

The 100 classes of CIFAR-100 are organized into 20 superclasses, providing a hierarchical structure to the labels, as shown in Fig 2. Each image is assigned both a ”fine” label indicating its specific class and a ”coarse” label indicating its superclass. This hierarchical labeling scheme introduces additional complexity to the dataset, enabling more nuanced and sophisticated learning tasks. In this report, however, we will only be utilizing the 100 fine labels as our target classes.

Data Preprocessing

Understanding the data and the network architecture is essential before starting the training process. This is because the quality and relevance of the input data directly impact the performance of the model. The CIFAR-100 training data from torchvision contains 50,000 images, which we'll randomly divide into a new training and validation set containing 40,000 and 10,000 images, respectively, using the following lines of code:

```
from torch.utils.data import DataLoader
from torch.utils.data SubsetRandomSampler
from torchvision import datasets, transforms

def get_train_valid_loader(dataset_dir, batch_size,
shuffle, seed, save_images=False, mean_std = None,
valid_size=0.2, pin_memory=True, num_workers=2):

    # Normalization Parameters
    if mean_std is not None:
        mean, std = mean_std
    else:
        mean = (0.5, 0.5, 0.5)
        std = (0.5, 0.5, 0.5)

    print(f'Mean: {mean}, Std: {std}')

    # Transform Parameters
    transform = transforms.Compose([
        transforms.RandomHorizontalFlip(p=0.5),
        transforms.RandomCrop(32, padding=4),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ])

    print(dataset_dir)

    # Load dataset from torchvision
    train_dataset = datasets.CIFAR100(root=dataset_dir,
    train=True, download=save_images,
    transform=transform)

    num_train = len(train_dataset)

    print(num_train)

    indices = list(range(num_train))
    split = int(np.floor(valid_size * num_train))

    if shuffle:
        np.random.seed(seed) # <--- set seed to 0
        np.random.shuffle(indices)

    # Split, sample, and instantiate the Dataloader
    train_idx = indices[split:]
    valid_idx = indices[:split]

    train_sampler = SubsetRandomSampler(train_idx)
    valid_sampler = SubsetRandomSampler(valid_idx)

    train_loader = DataLoader(train_dataset,
                              batch_size=batch_size,
                              sampler=train_sampler,
                              num_workers=num_workers,
                              pin_memory=pin_memory)
    valid_loader = DataLoader(train_dataset,
                              batch_size=batch_size,
                              sampler=valid_sampler,
                              num_workers=num_workers,
                              pin_memory=pin_memory)

    return train_loader, valid_loader

#Files already downloaded and verified
#Number of training samples: 40000
#Number of validation samples: 10000
```

Table 1 shows the proportion of each class in the new training dataset. We can see that the class distribution is relatively balanced around 1%, which is expected from the CIFAR-100 dataset. This balanced distribution is beneficial as it prevents the model from developing a bias towards any particular class during training.

The next step in our preprocessing pipeline is to calculate the mean and standard deviation of each color channel on the training set. These values will be used to normalize the training data. The calculated values are as follows:

```
Mean: tensor([0.5070, 0.4865, 0.4409])
Std: tensor([0.2673, 0.2564, 0.2761])
```

Next, we perform standard data augmentation techniques such as random horizontal flip with a probability of 0.5, random cropping with a 4-pixel padding and a patch size of 32-by-32, and normalization using the calculated mean and standard deviation values for each channel. These settings can easily be implemented in using the transforms.Compose() function of torchvision, as shown below:

```
...
# Iterate over the entire dataset to
# calculate mean and std (run once)
mean, std = get_data_mean_std(args.dataset_dir,
                              args.batch_size,
                              True,
                              args.seed)

...

# Transform Parameters
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
...
```

Table 1: The proportion of each class in the new training set

Class	Proportion	Class	Proportion	Class	Proportion	Class	Proportion
0	0.94%	25	0.99%	50	1.00%	75	0.97%
1	1.03%	26	1.00%	51	0.98%	76	1.00%
2	1.00%	27	0.96%	52	0.99%	77	1.01%
3	1.01%	28	1.00%	53	1.00%	78	1.01%
4	1.02%	29	1.03%	54	1.00%	79	1.00%
5	0.98%	30	1.02%	55	0.99%	80	0.99%
6	1.03%	31	1.01%	56	0.96%	81	1.02%
7	1.02%	32	1.00%	57	1.00%	82	0.99%
8	1.00%	33	1.01%	58	1.01%	83	1.01%
9	0.98%	34	0.96%	59	1.00%	84	0.95%
10	1.01%	35	1.00%	60	0.99%	85	1.00%
11	0.98%	36	1.04%	61	1.02%	86	0.99%
12	1.00%	37	1.00%	62	0.97%	87	1.03%
13	1.01%	38	0.99%	63	0.99%	88	1.00%
14	1.04%	39	1.00%	64	0.98%	89	1.03%
15	0.98%	40	0.98%	65	0.97%	90	0.99%
16	0.98%	41	1.03%	66	1.01%	91	0.99%
17	1.00%	42	1.01%	67	1.05%	92	0.99%
18	0.98%	43	1.03%	68	1.01%	93	1.03%
19	1.02%	44	1.03%	69	1.01%	94	0.98%
20	1.01%	45	1.04%	70	0.97%	95	0.98%
21	1.03%	46	0.98%	71	0.99%	96	1.02%
22	0.99%	47	1.00%	72	1.00%	97	0.97%
23	0.98%	48	0.99%	73	1.02%	98	1.04%
24	0.98%	49	0.99%	74	0.99%	99	0.99%

Tuning Hyperparameters

Striking a balance between optimization and regularization plays a pivotal role in the success of any deep learning task. Optimization, governed by, among other things, the initial learning rate and the learning rate scheduling, is concerned with the model's ability to learn from data and minimize training errors. On the other hand, regularization, influenced by factors such as, among other things, weight decay and data augmentation, aims to prevent overfitting and enhance the model's generalization capabilities. The hyperparameters controlling these aspects are critical for the performance of deep neural networks, which will be explored in this section. Each subsection offers insights into their roles, their impact on the model's learning process, and strategies for their effective tuning.

Initial Learning Rate

The first hyperparameter that we will explore is the *intial learning rate*. All other training hyperparameters and configurations are fixed and are as follows:

- batch size set to 128
- number of epochs set to 15
- SGD optimizer with momentum set to 0.9
- Simple Data Augmentation as described in the **Data Preprocessing** section

In this experiment, we tested three different initial learning rates: 0.01, 0.05, and 0.5. The learning rate is a critical hyperparameter that determines the step size at each iteration while moving towards a minimum of a loss function. A smaller learning rate could cause the model to converge too slowly, while a larger learning rate could cause the model to skip the minimum and diverge. After running our model with these different learning rates, we are able to generate learning curves as shown in Fig. 3 to Fig. 5. From the figures, we can see that the validation loss and accuracy curves for all three settings yielded roughly similar results, with a learning rate of 0.05 having a only very slight lead. If we are keen on further tuning the learning rate, we can conduct further testing on more epochs to observe long term effects on convergence, as well as test more increments of learning rate values. But for now, by solely relying on the final validation loss and accuracy values generated, we can use 0.05 as a starting point for future experiments.

Learning Rate Schedule

The next hyperparameter that we will explore is *learning rate schedule*, particularly, using cosine annealing. Cosine annealing is a strategy where the learning rate decreases following a cosine function between an upper and lower bound over a predefined number of epochs (Brownlee 2024).

Mathematically, cosine annealing is defined as:

$$\eta_t = \eta^i_{min} + \frac{1}{2}(\eta^i_{max} - \eta^i_{min})\left(1 + cos\left(\frac{T_{cur}}{T_i}\pi\right)\right) \quad (1)$$

where η_t is the learning rate at epoch t, η^i_{min} and η^i_{max}

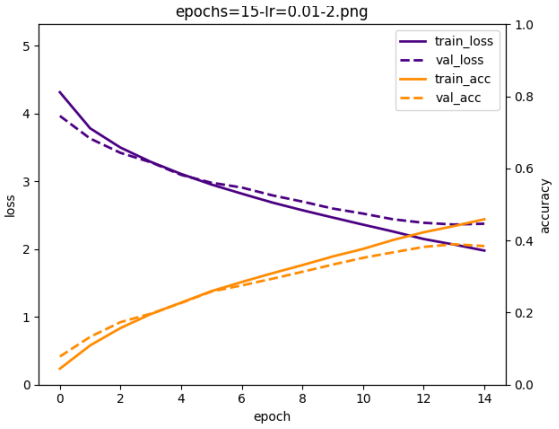

Figure 3: Set-up - learning rate = 0.01

.

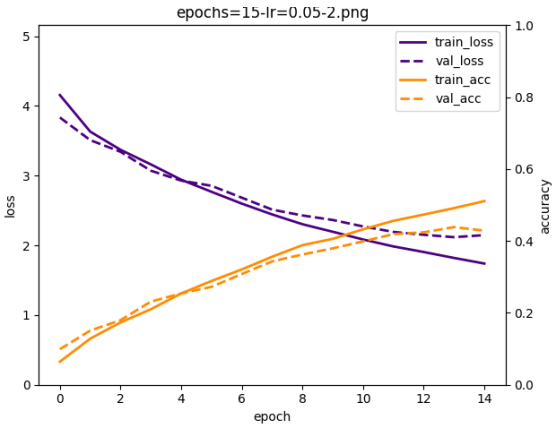

Figure 4: Set-up - learning rate = 0.05

.

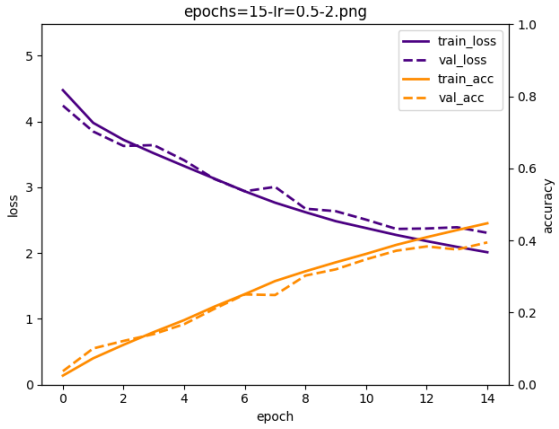

Figure 5: Set-up - learning rate = 0.5

.

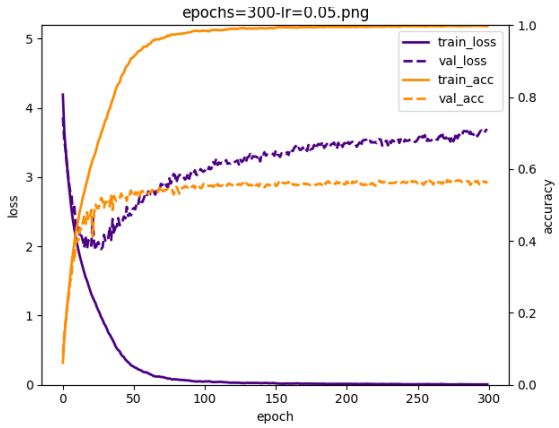

Figure 6: Set-up - lr=0.5, epoch=300

.

are the lower and upper bounds of the learning rate, respectively, and T is the total number of epochs. We know that $cos(x)$ produces values from +1 to -1 for values of x within the closed range $[0, \pi]$. From Eq. (1), we can see that this property is utilized using the ratio of the current epoch vs the total number of epochs, T_{cur}/T_i. This means that at Epoch 1, this ratio is close to zero, so the cosine term becomes +1, and continues as follows:

$$\begin{aligned} \eta_t &= \eta^i_{min} + \frac{1}{2}(\eta^i_{max} - \eta^i_{min})\,(1+1) \\ &= \eta^i_{min} + \eta^i_{max} - \eta^i_{min} \\ &= \eta^i_{max} \end{aligned} \tag{2}$$

Using the same equation for when the number of epochs equals T_i, we calculate the learning rate η_t to be equal to η^i_{min}. Here we see that the learning rate starts at η^i_{max} and gradually decreases to η^i_{min} following the cosine curve, before being reset to η^i_{max} for the next cycle. This approach allows the learning rate to decrease in a smooth and gradual manner, avoiding abrupt changes that could destabilize the learning process.

The experiment with the use of cosine annealing was conducted under two distinct settings: (1) 300 epochs with a constant learning rate of 0.05, and (2) 300 epochs with cosine annealing, starting with an initial learning rate of 0.05. The results are depicted in Fig. 6 and Fig. 7, respectively. A key point to consider is that, in terms of optimization, both cases were able to optimize as evidenced by the convergence of the training loss and accuracy. The application of learning rate scheduling, in this case, did not produce any striking advantage. However, in both cases, a discrepancy was observed in the trend of the validation loss and accuracy, which did not mirror the training metrics. This divergence is a clear indication of overfitting. In order to fix this, we need to apply regularization techniques, which will be explored next.

Weight Decay

The first regularization technique that we will apply is weight decay. Weight decay is a method used to prevent overfitting by directly subtracting $\eta\lambda w$, from the current w in each gradient descent update on $\mathcal{L}(w)$. By applying weight decay, we are effectively shrinking the weights to-

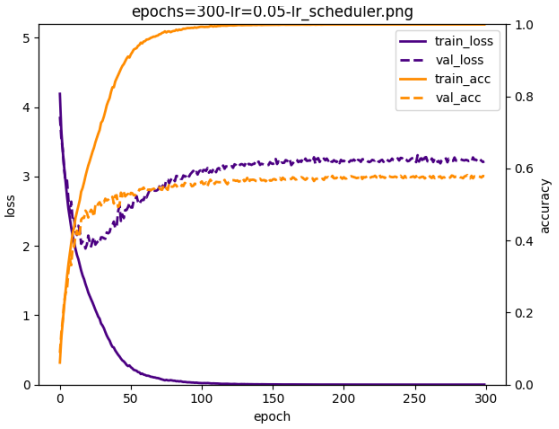

Figure 7: Set-up - lr=0.5, epoch=300, lr-scheduler

.

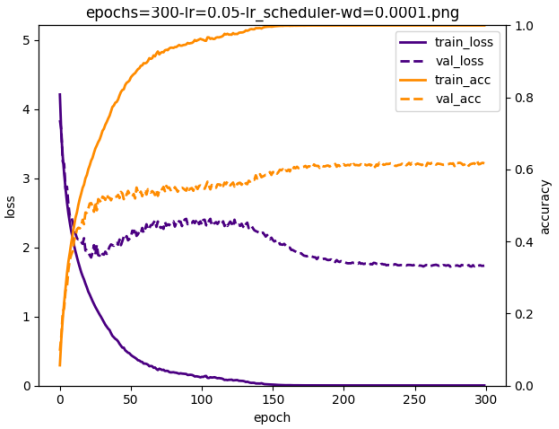

Figure 8: Set-up - lr=0.5, epoch=300, lr-scheduler, wd=1e-4

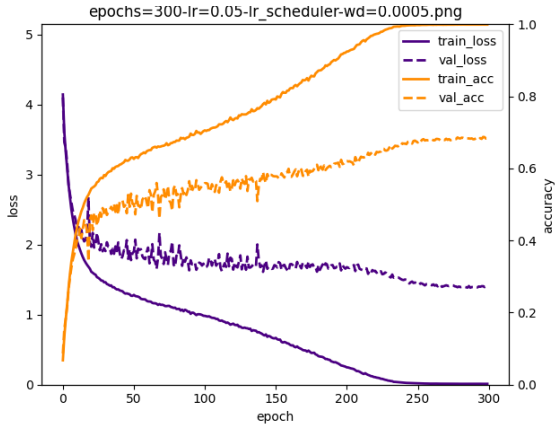

Figure 9: Set-up - lr=0.5, epoch=300, lr-scheduler, wd=5e-4

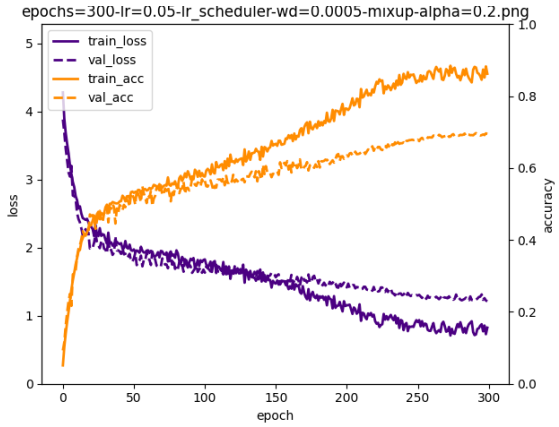

Figure 10: Set-up - lr=0.5, epoch=300, lr-scheduler, wd=5e-4, mixup, alpha=0.2

wards zero, making the model simpler and less likely to overfit to the training data. In this experiment, we tested two weight decay coefficients, $\lambda = 1 \times 10^{-4}$ and $\lambda = 5 \times 10^{-4}$ for 300 epochs, initial learning rate set to 0.05, and with cosine annealing. The training results are shown in Fig. 8 and Fig. 9, respectively.

The results illustrate the impact of regularization, specifically in terms of narrowing the gap between the training and validation curves. With a regularization parameter $\lambda = 1 \times 10^{-4}$, we observe a subtle shift that brings the training and validation curves closer together, as shown in Fig. 8. This effect is further amplified when λ is increased to 5×10^{-4}, as shown in Fig. 9. In both scenarios, we observe an enhancement in the validation curves, contrasted by a decline in the training curves, the magnitude of which is determined by λ. This observation demonstrates the trade-off between optimization and regularization.

Data Augmentation: Mixup

The second regularization technique that we will explore is Data Augmentation. In the course of the previous experiments, we have applied standard data augmentation techniques such as random horizontal flip and random cropping. In this section, we applied the MixUp augmentation technique, which generates virtual training examples through the convex combinations of pairs of inputs and their corresponding labels. This technique encourages the model to behave linearly in-between training examples, enhancing the generalization ability of the model. MixUp has been shown to provide a regularization effect and improve the performance of models on various datasets (Zhang et al. 2017).

For this experiment, the model is trained for 300 epochs with an initial learning rate of 0.05, with cosine annealing, weight decay $\lambda = 5 \times 10^{-4}$, and mixup with alpha = 0.2. The resulting training curves are shown in Fig. 10, and the final reported loss and accuracy values are presented in Table 2. For illustration, the probability density function (PDF) of the beta distribution parameterized by this alpha is displayed in Fig. 11.

Fig. 10 illustrates that the incorporation of MixUp, along with weight decay, significantly enhances the regularization effect. This is apparent from the substantial reduction in the gap between the training and validation curves. This pattern, again, shows the trade-off between optimization and regular-

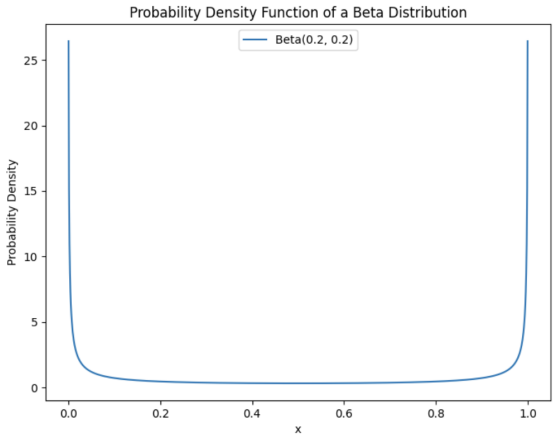

Figure 11: PDF of the Beta Distribution at alpha = 0.2

Table 2: Final Reported Loss and Accuracy Values

	Loss	**Accuracy**
Training	0.8248	0.8626
Validation	1.2117	0.6951

ization. Interestingly, the regularization effect in this experiment was so strong that the training curves did not converge within 300 epochs. However, it's important to note that a considerable gap still exists between the training and validation curves, indicating that our model is still overfitting. Therefore, additional regularization techniques may be considered for future improvements.

Finally, we evaluated our trained model on the hold-out test set and found that the model achieved a test accuracy of:

Test Accuracy = 60.29%

which is a considerable improvement compared to random chance (1%), but is, of course, still far from the performance of state-of-the-art solutions to CIFAR-100 (94 to 96%) (pwc 2024). This test accuracy is how we should expect the model to perform when deployed in actual applications.

The Takeaway Message

Throughout this assignment, I have gained a deeper understanding of the roles that optimization and regularization play in training deep learning models. The objective of optimization is to find the optimal parameters that minimize the model's loss function, thereby training the model to make the most accurate predictions possible. However, "learning" does not always align with optimization. For instance, a sufficiently deep neural network may have enough parameter space to effectively memorize the data, resulting in high training performance but falling short in terms of validation results (which may be the case in Fig. 6, 7, and 8). This issue is known as overfitting and is addressed using regularization. Regularization discourages the network from learning an overly complex model. This pushes the model to instead learn the underlying logic that governs the connection between the data and the ground truth labels, thereby enhancing the model's robustness to unseen data (as shown in Table 2 and Fig. 10). This "tug-of-war" between optimization and regularization is the foundation on which every deep learning model is built upon. In the future, I can apply these techniques to optimize my own models, ensuring they learn effectively from the data without overfitting.

References

2024. CIFAR-100 Benchmark (Image Classification) — Papers With Code.

Brownlee, J. 2024. Cosine Annealing Explained — Papers With Code.

Howard, A. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.

Krizhevsky, A.; and Hinton, G. 2009. Learning Multiple Layers of Features from Tiny Images. https://www.cs.toronto.edu/~kriz/cifar.html. Accessed: 2024-03-22.

Pujara, A. 2023. Image Classification With MobileNet.

Zhang, H.; Cisse, M.; Dauphin, Y. N.; and Lopez-Paz, D. 2017. mixup: Beyond Empirical Risk Minimization. *arXiv preprint arXiv:1710.09412*.