Effects of Hyperparameters Used in MobileNet on CIFAR-100

Tan Jie Heng Alfred
G2304193L

Abstract

Hyperparameters are parameters whose values are preset before the start of a model's training process. These values are not learned by the model. In this report, we will outline the effects of various hyperparameters used in MobileNet, which is a lightweight and efficient deep learning architecture. We tested the model on the CIFAR-100 dataset for image classification task. The hyperparameters tuned are: Learning rate, learning rate schedule, weight decay, and mix-up augmentation. We use the top-1 accuracy as our evaluation metrics to determine the best hyperparameter value. Finally, after finding the best hyperparameter values, we evaluate the performance of our model on a holdout test dataset.

Introduction

In any deep learning model, there are various parameters whose values determine the performance of the model on a task. Most of these parameters are learned by the model during training, but there are some that are pre-determined before the training of the model. The latter ones are called hyperparameters. Some examples of hyperparameters include the batch size, the learning rate, and weight decay. In this project, we will tune some of the hyperparameters in MobileNet, on the CIFAR-100 dataset for image classification task.

MobileNet

MobileNet is a convolutional neural network (CNN), but with some changes to allow it, to be more computationally efficient (Howard et al. 2017). In particular, the use of depthwise separable convolution allows for a significant reduction in the number of parameters (i.e., kernels) required, at the expense of the lack of interaction between output channels. However, with pointwise convolution, the output channels could interact through the help of 1×1 convolution. Despite its significantly smaller size, MobileNet only performs slightly worse than many larger CNNs, while being small enough to be operated on mobile phones.

CIFAR100

The CIFAR-100 dataset is a collection of images used to evaluate models for computer vision applications (Krizhevsky 2009). It consists of 60,000 RGB images, each of resolution 32×32. The images are divided evenly into 100 classes, each with 600 images. Each class is then further grouped into 20 superclasses (e.g., under the 'flower' superclass, there are 'orchids', 'roses', 'poppies', 'sunflower', and 'tulips'). In terms of training-test split, each class is divided into 500 training images and 100 testing images. Lastly, each image is labelled with both the class and superclass it belongs to. Typically, the dataset is used as a benchmark dataset for image classification and recognition task. In this project, we used the dataset for image classification task.

Hyperparameters

The hyperparameters that we studied in this project are the learning rate, the presence of a learning rate schedule, the weight decay, and use of mixup data augmentation. For each of the hyperparameter, we conducted comparative experiments to test their influence on a training set and a validation set. There are other hyperparameters within the MobileNet model such as the batch size and momentum, but these hyperparameters were held constant throughout all the experiments.

Many of the hyperparameters we used are either for more effective training (i.e., faster convergence) or for regularization (i.e., prevent overfitting). We want to introduce regularization into the model as it tends to optimize towards the training set, which is not representative of the underlying distribution. Hence, the regularization methods used in this project introduce soft constraints so that the model can generalize better.

In the next section, we will discuss the preprocessing done in preparation for the comparative experiments. Then, in each of the subsequent section, we will discuss the findings from the comparative experiments from each of the hyperparameter. Within each section, we will first discuss the significance of the hyperparameter, then we will share the training loss and accuracy, and validation loss and accuracy obtained while running the experiments. Finally, we will explore the significance of our findings and choose the best value to be used for the hyperparameter. Finally, after all the comparative experiments, we pick the best hyperparameter, and test our model on a holdout test set.

Listing 1: `train_val_split`

```
train_size = int(0.8*len(cifar100)) # cifar100 is variable pointing to the CIFAR100 training set

val_size = len(cifar100) - train_size

torch.manual_seed(seed) #seed = 0

train_dataset, val_dataset = random_split(cifar100, [train_size, val_size])
```

Data preprocessing

Before the training of a model, we would have to perform some data preprocessing. In this project, we first import the CIFAR100 training set consisting of 50,000 images. Then, we randomly divide this dataset into a new training set and a validation set using an 8:2 split (see Listing 1). The random seed for this split is set to 0, ensuring that the random split will always produce the same set of training and validation set. The proportion of each class is illustrated in the bar graph in Figure 1, where the horizontal axis represents the class number, and the vertical axis shows the proportion of the corresponding class number with respect to the new training set. We can see that the proportion of all class is roughly 1%. Hence, our training will not be too biased towards a certain class.

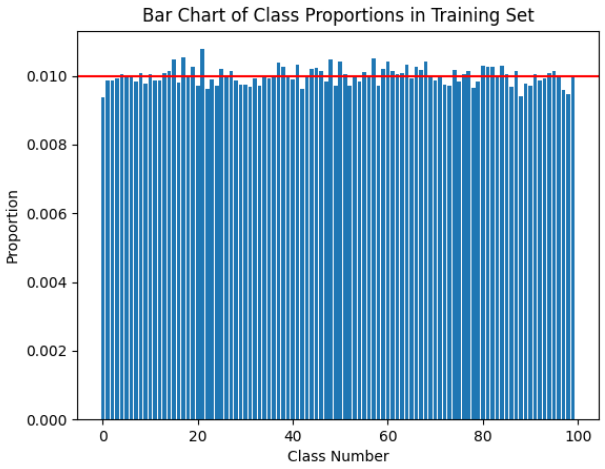

Figure 1: Proportion of each class with respect to new training set. Red line indicates the mean proportion of 0.010, which acts as a reference

After the split, we compute the mean and standard deviation of the images in the training set across each colour channel (RGB). The mean of the `[R, G, B]` channels are `[0.5068, 0.4861, 0.4403]`, and the standard deviation of the `[R, G, B]` channels are `[0.5068, 0.4861, 0.4403]`. Then, these values are used to whiten the training, validation, and test images, such that the training images will have a mean of 0 and standard deviation of 1 across all the channels. The normalization process makes the training of the model more efficient.

Listing 2: `transform_training_set`

```
train_transforms = transform.Compose([
    transforms.RandomHorizontalflip(0.5),
    transforms.Pad(4),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std)
])
```

Prior to the normalization of the training set, we also perform random horizontal flip and cropping. For the former, each of the image has a probability of 0.5 of being flipped horizontally. As for the latter, we first pad each of the image with 4-pixel, before randomly cropping a 32×32 patch from the padded image. The codes to do this are shown in Listing 2.

Once the preprocessing steps are done, we will perform the hyperparameter tuning. We will primarily tune the hyperparameter against the validation set, as the images in the validation set are not trained on the model, hence would appear novel to the model, for fixed hyperparameters.

For the subsequent comparative experiments, we will fix the batch size to be 128, use stochastic gradient descent as our optimizer, with momentum set to 0.9.

Learning Rate

In this section, we will investigate the effect of the initial learning rate η. The learning rate of a model is a hyperparameter that determines the extent to which the weights update per iteration. Mathematically, we have,

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \frac{\partial L}{\partial \mathbf{w}}\bigg|_{\mathbf{w}_{t-1}} \tag{1}$$

where $\mathbf{w}$ are weight parameters in the model, L is the loss function, and $\frac{\partial L}{\partial \mathbf{w}}\Big|_{\mathbf{w}_{t-1}}$ is the update (i.e., gradient descent) to $\mathbf{w}_t$ using the value $\mathbf{w}_{t-1}$. While a small learning rate results in slower convergence towards a local minima of the loss function, a larger learning rate may result in other problems such as overshooting and oscillation. Our aim here is to pick a learning rate that avoids both of these issues as much as possible.

For this section, we trained the network with three different learning rates, $\eta = 0.01, 0.05$ and 0.5, for 15 epochs each. Every other hyperparameters are kept the same during these experiments. To evaluate their performances, the metrics we used are the training loss and accuracy, and the validation loss and accuracy.

Here, we use the top-1 accuracy, which measures whether the best guess from the model (i.e., the class with highest probability) matches the actual class of the image. The results from these experiments are shown in Table 1, and Figure 2.

Learning Rate, η	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
0.01	2.01	0.450	2.36	0.395
0.05	**1.68**	**0.524**	**1.99**	**0.467**
0.5	1.94	0.462	2.14	0.431

Table 1: Final training loss and accuracy, and final validation loss and accuracy, after 300 epochs, corrected to 3 significant figures.

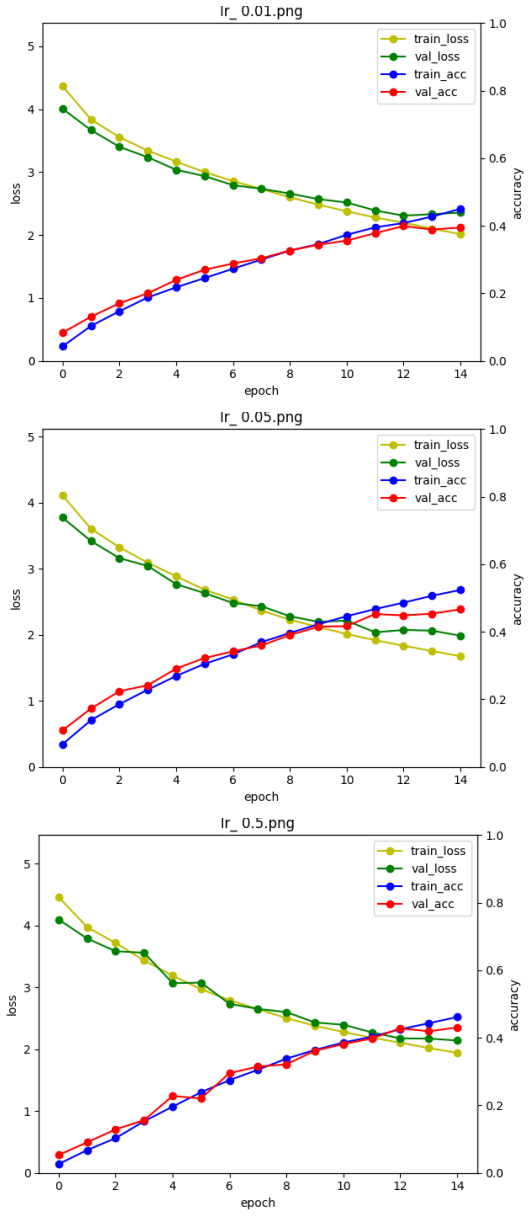

Figure 2: Graphs of training results with different learning rate. From the top, the corresponding learning rates are $\eta = 0.01, 0.05$, and 0.5.

It can be seen that $\eta = 0.05$ perform the best amongst the three learning rates, across all the evaluation metrics, with the lowest losses and highest accuracies. On the other hand, we can see that $\eta = 0.01$ has the highest losses and lowest accuracies. This is because at each iteration, the weights update very little, resulting in very slow convergence. Indeed, from Figure 2, the slope of `train_loss` for $\eta = 0.01$ is the gentlest.

For $\eta = 0.5$, we see that the model does not perform as well as when $\eta = 0.05$. This is because we are using the gradient as the 'direction' for our update. Since gradient is a local property, taking larger updates with larger learning rate would break this local property. Therefore, even when the training loss is decreasing, the parameters may be overshooting the local minima, resulting in poorer optimization.

From here, we will use $\eta = 0.05$ for subsequent experiments.

Learning rate schedule

Learning rate schedules are used to reduce the learning rates, η, as training progresses. This is done with the assumption that, as the model is being trained, the parameters are closer to the local minima than before and would require smaller updates to continue this convergence. In this section, we will be experimenting with cosine annealing as our learning rate schedule.

Mathematically, cosine annealing follows this equation in updating the learning rate.

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + \cos\frac{\pi t}{T}), \qquad (2)$$

where η_{max} and η_{min} are the initial and final learning rate that we have to set, T is the total number of epochs to be run, and t is the current epoch. Intuitively, our learning rate will decrease from one epoch to another (because cosine is monotonically decreasing in the domain $[0, \pi]$), from η_{max} to η_{min}. This rate of decrease is modelled after the first half-cycle of a cosine curve.

Here, we are using $T = 300$ and $\eta \in [0, 0.05]$, hence, the learning rate for our model will following the equation:

$$\eta_t = \frac{1}{2}(0.05)(1 + \cos\frac{\pi t}{300}) = 0.025(1 + \cos\frac{\pi t}{300}) \qquad (3)$$

Figure 3 shows how the learning rate used varies with epoch, t, for our experiment.

From Figure 3, we can see that the learning rate decreases the quickest in the middle of the training process (i.e., 150 epochs), and the slowest near the start and end of the training.

We conducted two experiments – one where the model is trained with a constant learning rate (i.e., without any schedule) and one where it uses cosine annealing. We use an initial learning rate of $\eta = 0.05$, found to be the best in the previous section, and trained the models for 300 epochs each. The results are shown in Table 2 and Figure 4

The model with cosine annealing performed better in all the evaluation metrics, with higher accuracies and lower losses. The main difference between the two models, as

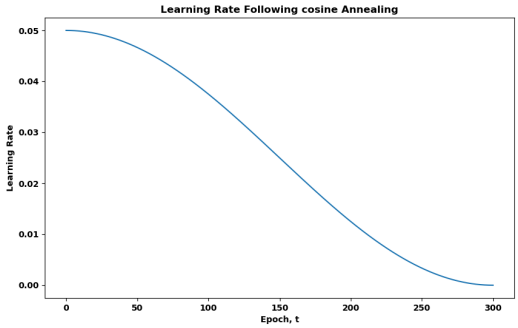

Figure 3: Learning rate under cosine annealing

Schedule Method	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
Constant η	0.00686	0.998	3.75	0.561
Cosine Annealing	**0.00234**	**0.999**	**3.67**	**0.626**

Table 2: Final training loss and accuracy, and final validation loss and accuracy, after 300 epochs, corrected to 3 significant figures.

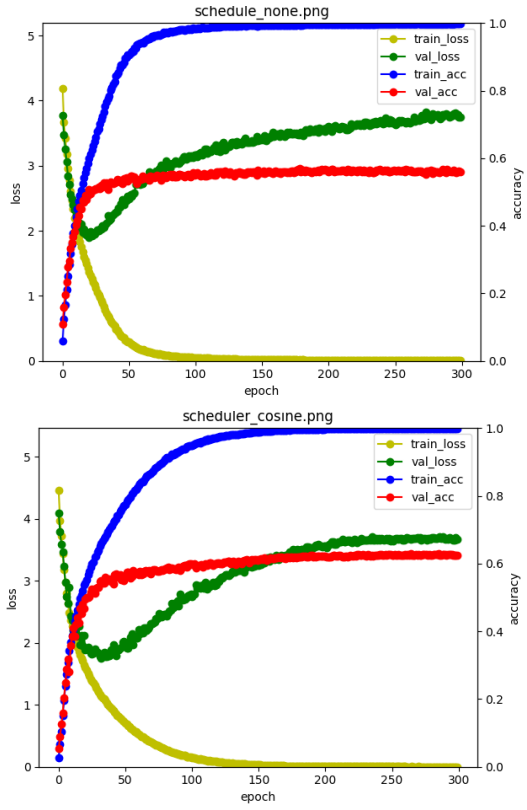

Figure 4: Graphs of training results with different learning rate schedule. From the top, the corresponding schedules are Constant η (i.e., no schedule) and Cosine Annealing

outlined above, is the change (or lack thereof) in learning rate value as training progresses. This difference is reflected in the gradients of `train_loss`, `val_loss` and `train_acc`, where the model with cosine annealing exhibits smoother gradients compared to the model with a constant learning rate.

While the constant learning rate model converges faster towards a lower training loss, it also overfits more severely to the training data. This is evident in the eventual increase in `val_loss` after roughly 25 epochs. Although overfitting is present in the model trained with cosine annealing, it is less severe (i.e., gentler increase in `val_loss` and a smaller final validation loss). Furthermore, the overfitting happens less quickly at about epoch 40. This is because the model with cosine annealing puts less importance on the training images as training progresses, since the update to the weights per iteration is decreasing with smaller η. This leads to the model not putting too much emphasis on the training images, hence modelling less of the the intrinsic noise present in the training dataset as compared to the constant learning rate model. It therefore learns a more general representation for the actual underlying distribution, and would perform better on unseen images, which is evident from the validation accuracy. As such, we will use cosine annealing for the remaining experiments.

However, both models still suffer from rather severe overfitting, as evident from the huge gap between training validation losses, and training and validation accuracies. To better generalize, we will explore the use of weight decay.

Weight decay

Weight decay is a regularization technique that aims to overcome overfitting by adding a penalty term to the loss function. The penalty applied is proportionate to the L2 norm of the weights. This is equivalent to subtracting a term proportional to $\mathbf{w}_{t-1}$ in the gradient descent update step:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta \frac{\partial L}{\partial \mathbf{w}}\bigg|_{\mathbf{w}_{t-1}} - \eta\lambda\mathbf{w}_{t-1}, \tag{4}$$

where λ is the weight decay coefficient.

From the equation, it can be seen that the $-\eta\lambda\mathbf{w}_{t-1}$ contributes to the decaying effect of update to $\mathbf{w}_t$. Moreover, weights are 'encouraged' to have smaller L2 norm, as larger values (in each entry) of $\mathbf{w}_{t-1}$ also lead to proportionally larger penalization by the $-\eta\lambda\mathbf{w}_{t-1}$ term. This controls the 'size' (i.e., L2 norm) of the weights, preventing the model from representing a larger range of functions, hence addressing overfitting.

In this section, we experimented with two different weight decay coefficients, $\lambda = 5 \times 10^{-4}$ and 1×10^{-4}. The results are shown in Table 3 and Figure 5.

As compared to previous experiments, the validation losses do not seem to increase indefinitely after training for more than 50 epochs. Instead, with weight decay, both models show an eventual decrease in validation loss, albeit with a small increase in validation loss during training for $\lambda = 1 \times 10^{-4}$.

Weight Decay λ	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
1×10^{-4}	0.00341	1.00	1.69	0.625
5×10^{-4}	**0.0119**	**1.00**	**1.39**	**0.683**

Table 3: Final training loss and accuracy, and final validation loss and accuracy, after 300 epochs, corrected to 3 significant figures.

Although when $\lambda = 1 \times 10^{-4}$, the model achieves a lower training loss, the lower validation loss and higher validation accuracy of the model with $\lambda = 5 \times 10^{-4}$ suggests that the latter model is better at overcoming the overfitting problem. An increasing gap between validation and training losses (and accuracies) suggests that the model is only performing better against the training dataset, but is not general enough to handle untrained data as well. From Figure 5, the gap between validation and training losses (and accuracies) are smaller in the model with $\lambda = 5 \times 10^{-4}$, compared to when $\lambda = 1 \times 10^{-4}$. This means that the overfitting problem is still very prominent when $\lambda = 1 \times 10^{-4}$, hence we will use $\lambda = 5 \times 10^{-4}$ in subsequent experiments to apply a stronger penalization to weights with larger L2 norms.

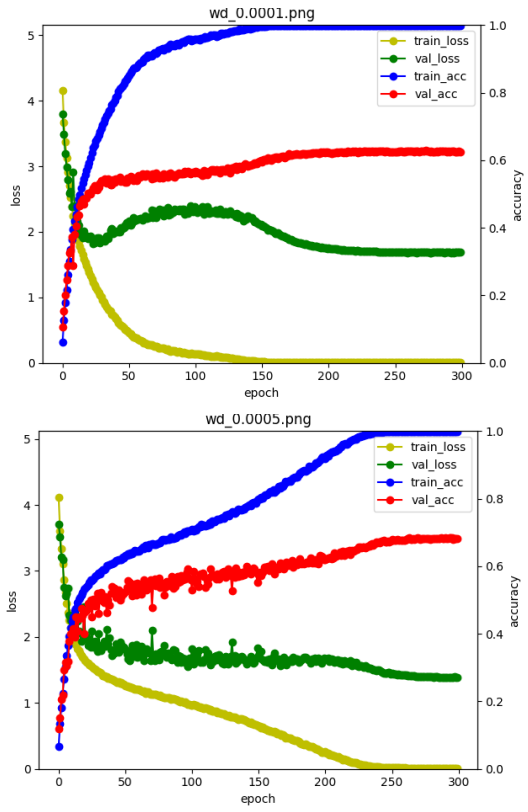

Figure 5: Graphs of training results with different learning rate schedule. From the top, the corresponding schedules are Constant η (i.e., no schedule) and Cosine Annealing

Mixup augmentation

Finally, we experimented with mixup, which is a data augmentation technique to perform regularization and prevent overfitting.

In mixup augmentation, we attempt to create unseen training images from our existing training dataset. Let x_1 and x_2 be two images from our training dataset, and y_1 and y_2 be their corresponding classes. Also, let $\lambda \sim \text{Beta}(\alpha, \beta)$ be a parameter sampled from the Beta distribution (parametrized by α and β). Then, we create a new augmented image $\tilde{x}$ and its corresponding label $\tilde{y}$ by:

$$\begin{cases} \tilde{x} = \lambda x_1 + (1-\lambda)x_2, \\ \tilde{y} = \lambda y_1 + (1-\lambda)y_2 \end{cases} \tag{5}$$

This can be conceptualized as an 'interpolation' between x_1 and x_2.

In our experiment, we used the parameters $\alpha = 0.2 = \beta$ for the Beta distribution. Therefore, λ will be sampled from the distribution $\text{Beta}(0.2, 0.2)$, illustrated in Figure 6.

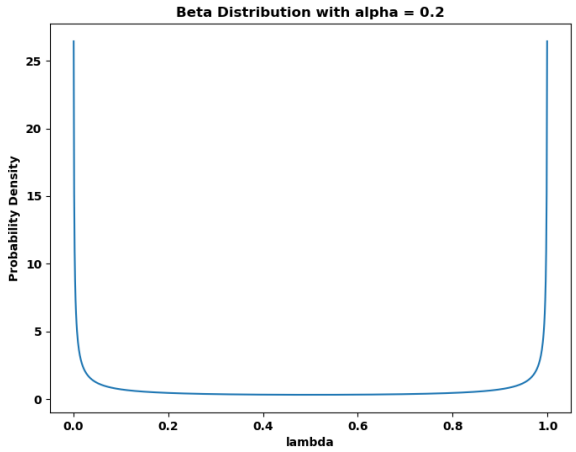

Figure 6: Beta distribution, with $\alpha = 0.2 = \beta$. The area under the graph between the values $\lambda = b$ and $\lambda = a$ gives $P(a < \lambda < b)$

Due to the Beta distribution and the values of α and β, our augmented image tends to be closer to either x_1 or x_2, and so would the augmented label. Since we do not know the underlying distribution of our data, having the 'interpolation' to be as close as existing data makes our augmented images more likely to be a datapoint sampled from the unknown distribution.

In this section, we experimented with mixup augmentation and the results of this model with the previous model, with weight decay of 5×10^{-4} (see Figure 5, bottom graphs). Both models are similar except for the use of mixup. For the first model, we continuously apply different mixup on the original training images during each epoch, and ran it for 300 epochs. The results are shown in Table 4 and Figure 7.

In the presence of mixup augmentation, the final training loss has increased quite substantially, which results in

Mixup	Training Loss	Training Accuracy	Validation Loss	Validation Accuracy
Present	**0.835**	**0.862**	**1.21**	**0.698**
Absent	0.0119	1.00	1.39	0.683

Table 4: Final training loss and accuracy, and final validation loss and accuracy, after 300 epochs, corrected to 3 significant figures.

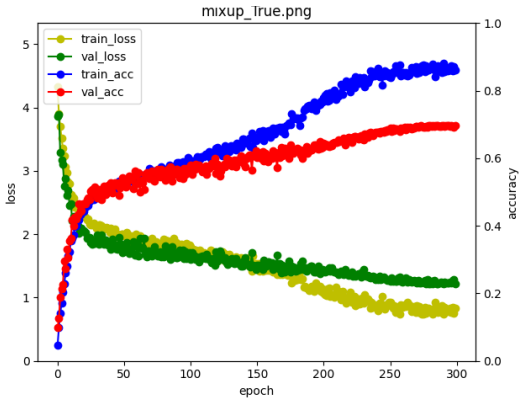

Figure 7: Graphs of training results with mixup augmentation. The graphs of training results without mixup augmentation can be found in Figure 5 as the bottom graphs.

a decrease in training accuracy. In fact, from Figure 7, the training with mixup augmentation results in some jittering in the training loss during the training process. This is a result of the training images being changed ever so slightly from one epoch to the next by the mixup. It then prevents the model from simply overfitting to a fixed training set. This also forces the model to learn weights that generalize well to a wider variety of images, as the training images are different in each epoch. Ultimately, these result in a lower validation loss and higher validation accuracy, as the parameters in this model are able to generalize better towards the untrained images in the validation set. Hence, we choose the model with mixup augmentation.

Performance on holdout test set

Lastly, to test the performance of our model, we use a holdout test set, obtained from the CIFAR100 test dataset, on our model. These data are truly new to our model, as they were not used in the training of the parameters, as well as the tuning of the hyperparameters. Hence, the performance of our model on this dataset will be indicative of how our model will perform on new data. Upon running the inference on the test set, our model produced a loss of 0.0019 accuracy of 0.697. The accuracy is very similar to validation accuracy we obtained above.

Conclusion

Hyperparameters tuning is an essential step of training a generalized model. Many of these hyperparameters discussed either result in faster convergence or perform regularization on the model. These are important in ensuring a more efficient training and a more generalized model that is able to perform well with data that was not seen previously. To that end, we tuned our hyperparameters with the main focus of optimizing for validation loss and accuracy, as these metrics give a better proxy regarding how the model would perform to data that it was not trained on. From our experiments, we found that an initial learning rate of 0.05 that decreases during training with cosine annealing, and the presence of weight decay of 5×10^{-4} and mixup augmentation, allows the model to train efficiently and generalize better to unseen data. In fact, when tested with a test set, the model was able to achieve a top-1 accuracy of 0.697.

References

Howard, A. G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; and Adam, H. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 1312–1320.

Krizhevsky, A. 2009. CIFAR-10 and CIFAR-100 datasets. https://www.cs.toronto.edu/~kriz/cifar.html.