International Baccalaureate Diploma Program

Computer Science Extended Essay

ResNet: A Comparative Study of Depth and Width in Neural Network Architectures across different datasets

To what extend does increasing width and depth affect the ResNet's performance (Accuracy, Training time, F-1 score, Marginal gains) across various datasets?

3820 Words

Contents

1 Introduction

Since the groundbreaking success of the convolutional neural network - AlexNet in 2012, CNNs have experienced exponential growth. Consequently, various new models have emerged every year in the ILSVRC competition to surpass accuracy records. In 2014, Christian Szegedy et al. introduced VGG and GoogleNet, significantly enhancing the model's feature representation capability and learning capacity by creating deeper network architectures. However, deeper models still faced a fundamental challenge - the vanishing gradient problem. This limitation stemmed from inherent design inadequacies in neural networks, impeding effective weight updates in deep neural networks and sometimes rendering networks entirely untrainable. In 2015, this issue is addressed by constructing the deep residual network ResNet using identity mapping by shortcut, successfully implementing networks with up to 1202 layers (He, Zhang, Ren, & Sun, 2016). However, deeper networks also imply longer training times. Since the objective of the ResNet paper was to construct deep networks, it needed to provide more discussion on performance. Subsequently, Wide Residual Network (WRN) was proposed, suggesting that wider and shallower networks outperform deeper networks (Zagoruyko & Komodakis, 2016). However, it did not provide any conclusion to the influence of the datasets to the networks' performance. Thus, two apparent pathways to enhancing neural network models are increasing width or depth. This raises a practical question: Which is more effective, increasing depth or width?

More concretely, it is possible to ask about the gains brought by increasing depth or width to the model's accuracy and to what extent in different scenarios. Can it improve marginal gains and F-1 scores? This paper will address these questions by conducting experimental studies with different widths and depths of ResNet family models on the CIFAR-10, FashionMNIST and GTSRB datasets.

2 Theoretical Background

2.1 ResNet Revisited

2.1.1 Gradient Vanishing

Gradient vanishing is a common issue encountered in deep networks, rooted in the intrinsic mechanism of neural networks. Neural networks update their parameters through backpropagation based on the chain rule in Calculus. This implies that during the gradient descent process, if the derivative of some parts of the activation function remains consistently less than 1, then as the network deepens, the gradient will exponentially decay, leading to the gradient vanishing (Amanatullah, 2023).

To further elaborate, consider a neural network with two layers. Let the input to the network be X, the weights of the first layer be $W^{[1]}$, the bias of the first layer be $b^{[1]}$, the activation function of the first layer be $g^{[1]}$, the output of the first layer be $A^{[1]}$, the weights of the second layer be $W^{[2]}$, the bias of the second layer be $b^{[2]}$, the activation function of the second layer be $g^{[2]}$, and the final output of the network be Y. Therefore, the mathematical equations can be expressed as follows:

$$\begin{aligned} Z^{[1]} &= W^{[1]}X + b^{[1]} \\ A^{[1]} &= g^{[1]}(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ Y &= g^{[2]}(Z^{[2]}) \end{aligned} \tag{1}$$

The *backpropagation* will compute the gradients of weight parameters $W^{[1]}$ with respect to the *Loss function* to update the values, which is denoted by L. The gradients computation is as follows:

$$\frac{\partial L}{\partial W^{[1]}} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial Z^{[2]}}\frac{\partial Z^{[2]}}{\partial A^{[1]}}\frac{\partial A^{[1]}}{\partial Z^{[1]}}\frac{\partial Z^{[1]}}{\partial W^{[1]}} \tag{2}$$

To update the weights, the *Gradient descent algorithm* is employed:

$$W^{[1]} = W^{[1]} - \alpha\frac{\partial L}{\partial W^{[1]}} \tag{3}$$

where α is the learning rate.

Evidently, when the value of $\frac{\partial L}{\partial Y}\frac{\partial Y}{\partial Z^{[2]}}$ becomes small enough, at which the fitting capacity of the second layer is strong enough leading to little difference between the predictions and actual values, the derivative would be directly transferred to the value of $\frac{\partial L}{\partial W^{[1]}}$ and make it small. As a result, based on the Gradient descent equation above, $W^{[1]}$ would receive little modification. This is just an illustration under the context of a two-layer neural network. As the network depth increases, it can be observed that the weight parameters closer to the input layer are updated more slowly.

2.1.2 Solution provided by ResNet

To avoid the gradient vanishing problems faced in deep convolutional neural networks, He and his teams introduced skip connections and designed residual blocks (He et al., 2016). The residual block is the building block of ResNet, and the illustration of the design is shown below:

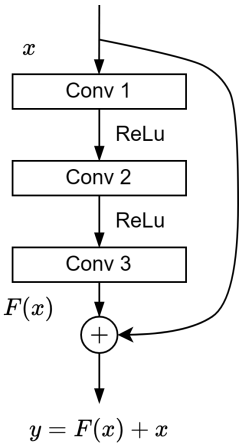


Figure 1: The building block of ResNet architecture, where x is the input, y is the output, $F(x)$ represents three convolutional layers.

To elaborate why this design could make a change, suppose a ResNet with output y'' consisting of two residual blocks, denoted by $f(X)$ and $g(X)$ respectively, where X is the input. According to the building block design above, the output of the network could be expressed as:

$$y'' = f(X) + g(f(X)) \tag{4}$$

Let $y = f(X)$, $y' = g(f(X))$. According to the chain rule, the gradient of the loss function L with respect to the weight parameter W of the first residual block is expressed as:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y''}(\frac{\partial y}{\partial W} + \frac{\partial y'}{\partial W}) \tag{5}$$

Even though the value of $\frac{\partial y'}{\partial W}$ is small enough, there's still a derivative of $\frac{\partial y}{\partial W}$ existed, therefore, avoid the gradient vanishing problems.

2.1.3 ResNet Architecture

Residual networks typically consist of 6 stages, with the first five stages handling convolution operations, while the final stage is a global average pooling layer. This paper only discusses the first

five stages. The first stage consists of a large kernel convolution and a max-pooling layer. The large kernel convolution aims to preserve original pixel information as much as possible, passing the result to the following four stages. These four stages consist of residual blocks with skip connections, each composed of n residual blocks, as illustrated in Figure 2.

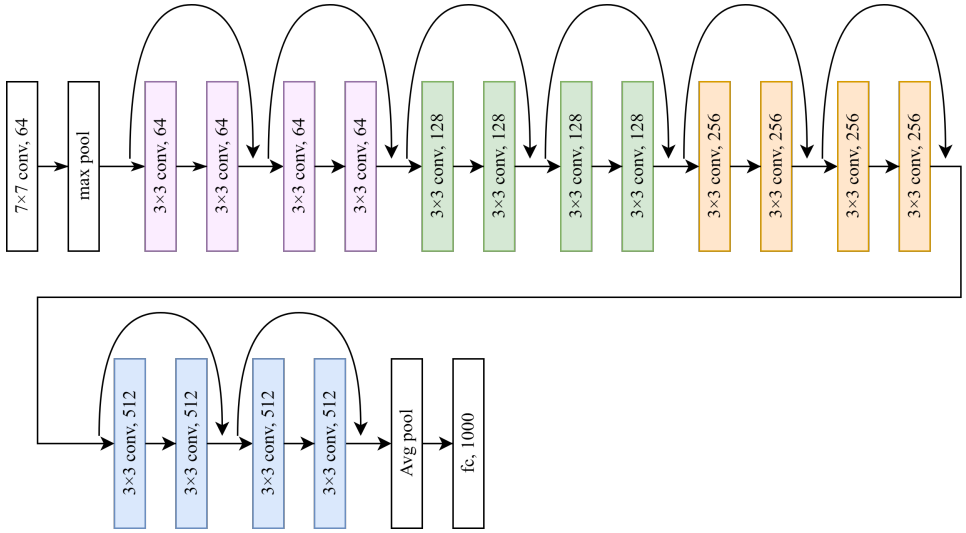


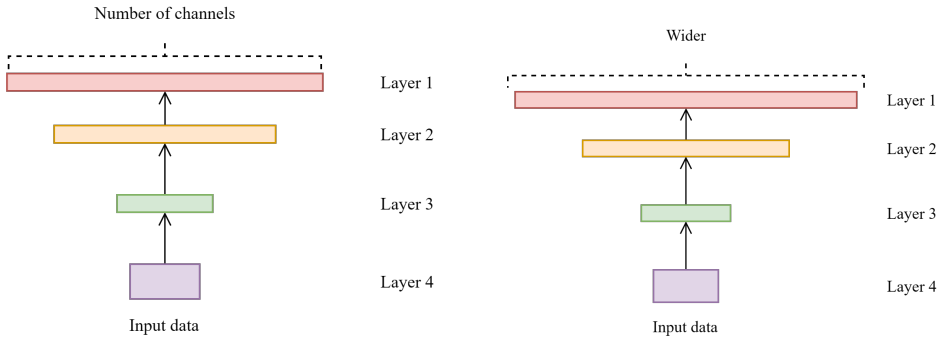
Figure 2: An Example of ResNet architecture diagram with 18 layers for 1000-class classification.

In the depicted diagram, the model employs four distinct colors to delineate the intermediate convolutional modules, housing eight residual blocks, each composed of two 3×3 convolutions. These four colors correspond to four stages, each characterized by various convolutional filters, namely $\{64, 128, 256, 512\}$. A higher count of filters in a stage enables the model to capture more features. The diagram illustrates a twofold filter increase for each successive stage, forming a pyramidal structure (Paoletti et al., 2019). Such architectural design facilitates the acquisition of more abstract features by the convolutional layers closer to the output, consequently enhancing the model's efficacy in image recognition tasks.

2.1.4 Width in ResNet

In convolutional neural networks, *width* refers to the capacity of each layer to process input data. One way to measure it is by the number of channels in the convolutional layer, also known as

the number of convolutional filters. Figure 3 illustrates the width of a convolutional neural network. In the residual network model shown in Figure 2, the width of the model ranges from $\{64, 128, 256, 512\}$. Increasing the width can enhance its learning capacity, allowing it to handle more information from input images. However, it also leads to longer training time and may even result in overfitting, where the model's performance on the training set significantly surpasses that on the validation set.



(a) Base Model (b) Increasing the width

Figure 3: Demonstration of width in convolutional neural networks.

2.1.5 Depth in ResNet

Depth refers to the number of layers in a convolutional neural network. The residual neural network depicted in Figure 1 has 17 convolutional layers and one fully connected layer, totaling 18 layers. Increasing the depth of a network is a widely accepted practice in the industry. Deeper models possess more robust capabilities to fit data and, therefore, can perform more complex tasks. In visual models, deeper architectures can extract more detailed and abstract features at different depths, leading to more accurate recognition tasks. However, increasing the depth comes with the cost of higher computational resources and longer training times. Notably, in the ResNet architecture, the authors introduced 1×1 convolutions to reduce the model's parameter count, thereby implementing a bottleneck building block design, as illustrated in Figure 4.

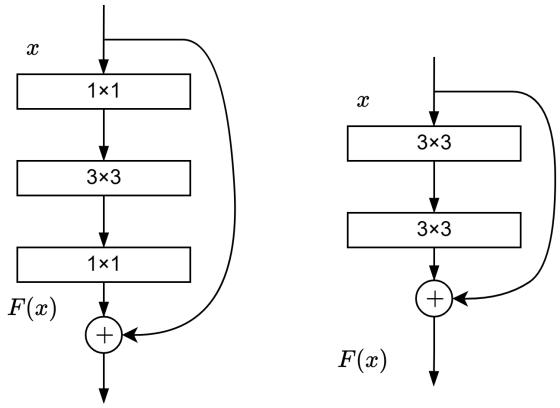


Figure 4: The building block of ResNet architecture in practice. Left: a conventional building block. Right: a bottleneck building block

2.1.6 Wider Or Deeper

In summary, it is worth noting that increasing the width of a neural network leads to a wider network, while increasing the depth results in a deeper network. Both of these strategies can potentially improve the model's learning capacity to a certain extent. However, the optimal choice between width and depth to maximize performance remains a conundrum. Nonetheless, it is imperative to first comprehend the degree to which each approach enhances the model's learning capabilities before making such determinations. This foundational understanding is indispensable for guiding any further adjustments and optimizations.

It is important to carry out empirical research to ascertain the effectiveness of widening or deepening a network in enhancing its learning capacity in differernt datasets since there's a great variance in performance while training the networks in different datasets. Besides, it should be noted that there exists a critical threshold beyond which the model becomes overwidened or overdeepened.

To this end, the empirical studies will encompass several components: Firstly, investigating how modifying the width and depth influences the model's performance across various datasets. Lastly, the maximum threshold for practical gains is determined.

3 Experiment Methodology

3.1 Experiment Setup

Experiment purpose is to understand the effects of width and depth on the model's performance in different situations and discover the threshold for practical gains. Reflecting this, the experiment setup would comprise the families of ResNets trained on the universal dataset GTSRB, CIFAR-10 and FashionMNIST, which represents datasets with different dimensions of images.

3.2 Datasets and Networks used

German Traffic Sign Recognition Benchmark (GTSRB) is an image classification dataset consists of 43 classes of traffic signs, splitting into 39209 training images and 12630 test images having the highest resolution among three datasets (Stallkamp, Schlipsing, Salmen, & Igel, 2012). The conventional ResNet architecture will be used in this dataset presented in Table 1.

Layer Name	ResNet-24	ResNet-50	ResNet-101
conv1	7×7,64,stride 2		
	3×3,max pool, stride 2		
conv2	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times3$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times3$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times3$
conv3	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times3$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times4$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times4$
conv4	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times3$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times6$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times23$
conv5	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times3$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times3$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times3$

Table 1: Conventional ResNet architectures of ResNet-24, ResNet-50 and ResNet-101.

CIFAR-10 is an established computer-vision dataset used for object recognition, which consists of 10 object classes, with 6000 images per class. Each image has dimensions of 32×32 pixels (Krizhevsky, 2009). However, the conventional ResNet architectures were built for the ImageNet

dataset, of which each image has dimensions of 224×224 pixels, which is completely different from the ones of CIFAR-10. Therefore, a variation of ResNet architectures will be used. In table 2, the adjusted architecture families including ResNet-20, ResNet-56 and ResNet-110 are shown, which would be used in the experiment. The first layer was changed to 3×3 convolutions with stride 1, channels of 16. For other three layers below, the numbers of channels was changed to $\{16, 32, 64\}$ respectively (Ruiz, 2018).

Layer Name	ResNet-20	ResNet-56	ResNet-110
conv1	3×3,16,stride 1	3×3,16,stride 1	3×3,16,stride 1
conv2	$\begin{bmatrix} 3\times3, 16 \\ 3\times3, 16 \end{bmatrix} \times 3$	$\begin{bmatrix} 3\times3, 16 \\ 3\times3, 16 \end{bmatrix} \times 9$	$\begin{bmatrix} 3\times3, 16 \\ 3\times3, 16 \end{bmatrix} \times 18$
conv3	$\begin{bmatrix} 3\times3, 32 \\ 3\times3, 32 \end{bmatrix} \times 3$	$\begin{bmatrix} 3\times3, 32 \\ 3\times3, 32 \end{bmatrix} \times 9$	$\begin{bmatrix} 3\times3, 32 \\ 3\times3, 32 \end{bmatrix} \times 18$
conv4	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 9$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 18$

Table 2: Architectures of adjusted ResNet families (ResNet-20, ResNet-56, ResNet-110).

The same model architecture series is also applicable to Fashion-MNIST. This dataset, the smallest among the three, consists of ten thousand grayscale images with dimensions of 28×28 pixels, with an input channel count of 1 (Han, Rasul, & Vollgraf, 2017).

3.2.1 Data Augmentation

Convolutional neural networks are robust, and augmenting them with increased width and depth may lead to overfitting, where the model's performance on the training set significantly surpasses that on the validation set. Thus, the experiments will employ data augmentation techniques to mitigate this issue.

Data augmentation involves modifying the original dataset to generate synthetic samples that still represent the same categories as the original samples, thereby preventing the model from simply memorizing the training data. This approach forces the model to generalize better and helps avoid

overfitting.

The experiments will utilize four data augmentation techniques:

1) Horizontal flipping

2) Vertical flipping

3) Rotation of 30 degrees

4) Random cropping

These techniques will be applied to the original dataset to generate augmented samples, thereby diversifying the training data and enhancing the model's ability to generalize to unseen examples.

3.2.2 Hyperparameters

In this experiment, a more stable optimization algorithm, stochastic gradient descent (SGD), will be utilized. For the Fashion-MNIST dataset, the initial learning rate is set to 0.1. The learning rate will decay by a factor of 0.1 at epochs 50, 90, and 110 until completing 150 epochs. The optimizer's momentum is set to 0.9, and the weight decay is 0.0001. The loss function employed is cross-entropy. For the CIFAR-10 dataset, learning rate decay will occur at epochs 120, 200, and 250 until 300 epochs are completed. Other parameters remain the same as for Fashion-MNIST. For the GTSRB dataset, learning rate decay will occur at epochs 20 until the completion of 40 epochs. The remaining parameters are consistent with the Fashion-MNIST setup.

3.2.3 Increasing the width and depth

The method to increase depth, as illustrated in Tables 1 and 2, involves directly increasing the number of residual blocks. The experiments will utilize the native ResNet models and select models with a depth difference of two-fold. For experiments on the CIFAR-10 and Fashion MNIST

datasets, ResNet-20, ResNet-56, and ResNet-110 will be employed. On the other hand, experiments on the GTSRB dataset will utilize ResNet-24, ResNet-50, and ResNet-101.

In the experiments on width, a width coefficient variable, denoted as ω, will be introduced. This variable will be multiplied by the number of channels in each convolutional layer within the residual blocks to increase the model's width, as shown in Table 3. The progression of this ascent will follow a similar pattern to that of increasing depth, i.e., doubling with each step. Therefore, the width coefficients array $W = \{2, 4\}$ could be derived.

Layer Name	ResNet-20	ResNet-56	ResNet-110
conv1	3×3,16,stride 1	3×3,16,stride 1	3×3,16,stride 1
conv2	$\begin{bmatrix} 3\times3, 16\omega \\ 3\times3, 16\omega \end{bmatrix} \times 3$	$\begin{bmatrix} 3\times3, 16\omega \\ 3\times3, 16\omega \end{bmatrix} \times 9$	$\begin{bmatrix} 3\times3, 16\omega \\ 3\times3, 16\omega \end{bmatrix} \times 18$
conv3	$\begin{bmatrix} 3\times3, 32\omega \\ 3\times3, 32\omega \end{bmatrix} \times 3$	$\begin{bmatrix} 3\times3, 32\omega \\ 3\times3, 32\omega \end{bmatrix} \times 9$	$\begin{bmatrix} 3\times3, 32\omega \\ 3\times3, 32\omega \end{bmatrix} \times 18$
conv4	$\begin{bmatrix} 3\times3, 64\omega \\ 3\times3, 64\omega \end{bmatrix} \times 3$	$\begin{bmatrix} 3\times3, 64\omega \\ 3\times3, 64\omega \end{bmatrix} \times 9$	$\begin{bmatrix} 3\times3, 64\omega \\ 3\times3, 64\omega \end{bmatrix} \times 18$

Table 3: Architectures of adjusted ResNet families with widen factor ω (ResNet-20, ResNet-56, ResNet-110).

Due to the relatively narrow width of the adjusted ResNet models, there exists considerable room for improvement. In order to further substantiate this assertion, an additional set of experiments will be conducted on the CIFAR-10 and FashionMNIST datasets, introducing a width coefficient of 4.

3.2.4 Measuring the performance of the models

The experiments will compare the training time, training accuracy, F-1 score and marginal gains. This comprehensive analysis aims to provide a holistic understanding of the impact of increasing depth and width on model performance.

Accuracy Accuracy is a metric that measures how much a model could correctly predicts the outcome.

Training time Training time refers to the total time to train the model given the number of epochs.

F-1 Score F-1 Score is a more synthesize index that assesses the prediction ability of a model by elaborating on its class-wise performance.

Marginal gain Marginal gains a measure of combination of accuracy and training time altogether, which ahs the equation below:

$$\text{Marginal gain} = \frac{\text{Percentage Testing Accuracy}}{\text{Total training time}} \tag{6}$$

4 Experiment Result and Analysis

4.1 CIFAR-10, FashionMNIST and Analysis

The comparison results of ResNet of different width and depth in CIFAR-10 and FashionMNIST datasets are shown in Table 4.

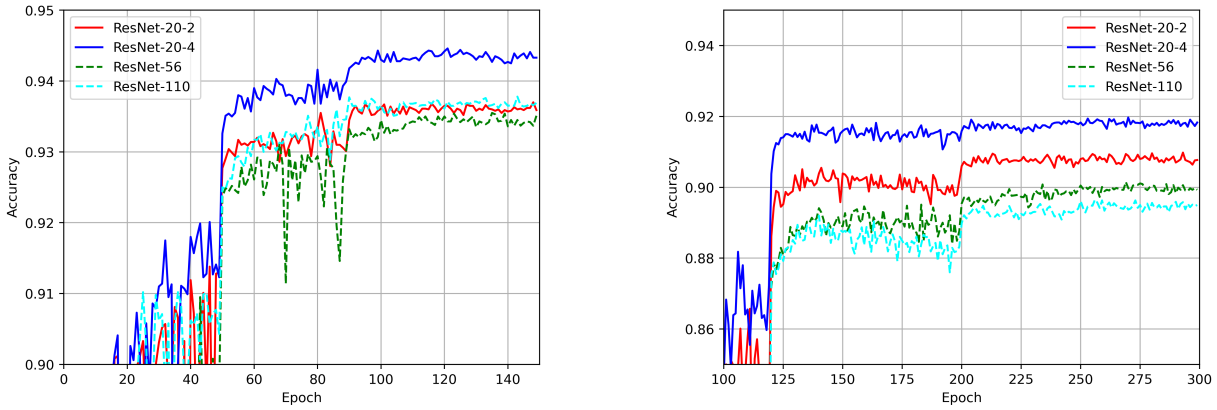
Model	Params	CIFAR-10 classification validation accuracy (%)	FashionMNIST classification validation accuracy (%)	CIFAR-10 Single Epoch Training Time (s)	FashionMNIST Single Epoch Training Time (s)
ResNet-20	0.27M	88.90	93.62	9.4	8.1
ResNet-56	0.85M	89.93	93.54	14.3	14.8
ResNet-110	1.73M	89.5	93.71	23.6	25.3
ResNet-20-2	1.07M	90.77	93.65	9.5	9.1
ResNet-20-4	4.29M	91.83	94.43	24.8	15.48
ResNet-20-8	17.1M	93.14	94.26	77.3	39.4

Table 4: The experiment results of ResNet with different width and depth in CIFAR-10 and FashionMNIST datasets. ResNet-20-2 means ResNet-20 plus width coefficient of 2, same as others.

The results in Table 4 show that wider models perform better than deeper models. ResNet-20-8 achieves the highest accuracy of 93.14% in CIFAR-10, which is 3.64% higher than ResNet-110. Even ResNet-20-4 outperforms ResNet-110 by 2.33%. Obviously, compared to increasing depth, increasing the width could gain more in validation accuracy in CIFAR-10 dataset.

In FashionMNIST, wider models still perform better than deeper models in overall. ResNet-20-4 achieves the highest accuracy of 94.43%, which is 0.72% higher than ResNet-110. ResNet-20-2 still outperforms ResNet-56 by 0.11%.

Figures 5(a) and 5(b), respectively, illustrate the learning curves of ResNet-56, ResNet-110, ResNet-20-2, and ResNet-20-44 on the validation sets of the CIFAR-10 and FashionMNIST datasets. On the CIFAR-10 dataset, the accuracy of the width models is higher than that of the depth models. Conversely, on the FashionMNIST dataset, which is smaller in scale and easier to train, the width models still maintain an advantage. However, in the case of ResNet-110, its overall trend indicates a higher accuracy compared to ResNet-20-2.



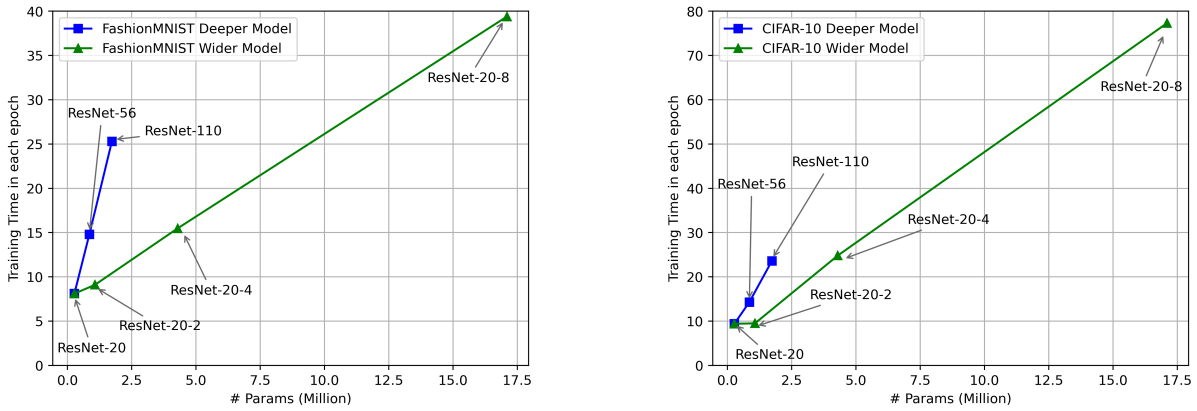
(a) Validation accuracy in FashionMNIST

(b) Validation accuracy in CIFAR-10

Figure 5: Validation accuracy in FashionMNIST and CIFAR-10 of ResNet-20-2, ResNet-20-4, ResNet-56, ResNet-110.

From the perspective of images alone, wide models have more advantages than deep models. While analysis in training time and number of params could provide more insights. According to the training results in Table 4, the parameter quantity of ResNet-20-4 is four times higher than that of the same depth-scaled ResNet-110.Normally, the larger the model's parameter quantity, the longer the training time. This hypothesis has only been proven on the CIFAR-10 dataset, where the single epoch training time of ResNet-20-4 is 4.83% higher than that of ResNet-110, which is not a propor-

tional increase in training time. However, the opposite results were obtained on the FashionMNIST dataset. The single epoch training time of ReNet-20-4 is 63.43% lower than that of ResNet-110, and the single epoch training time of ResNet-20-2 is also 62.64% lower than that of ResNet-56. These results are more intuitively presented in Figure 6. Considering that the size of a single image in both FashionMNIST and CIFAR-10 is relatively small, it can be concluded that wide and shallow models have more advantages both in time and accuracy on relatively small datasets than deep models.



(a) Training time in 1 epoch in FashionMNIST

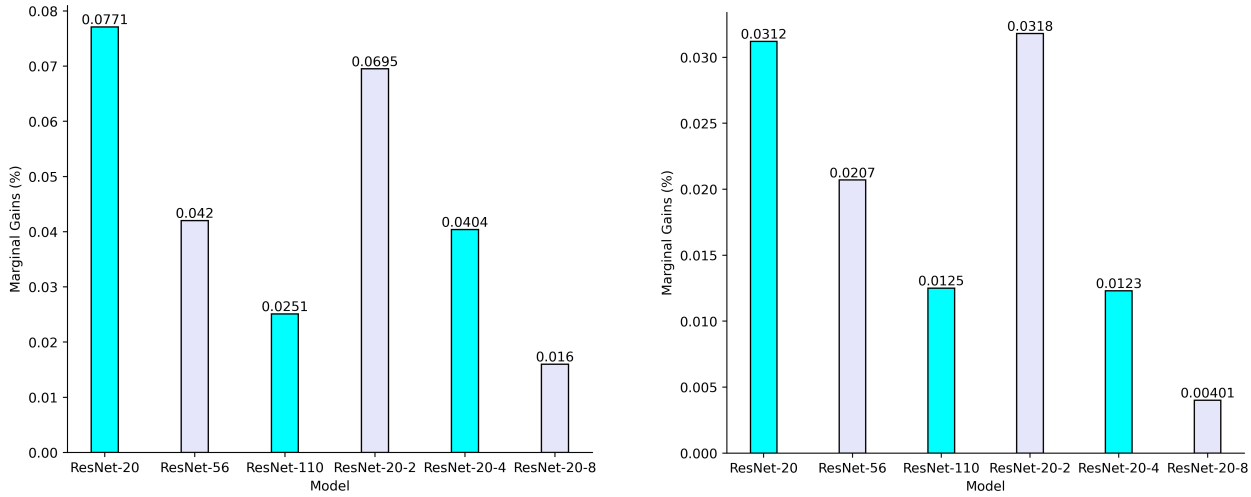
(b) Training time in 1 epoch in CIFAR-10

Figure 6: Training time vs number of parameters curves in 1 epoch for wide and deep models.

By contrast, the previous assumption (More parameters, more training time) holds for the ResNet-20-8 model, whose parameter quantity is four times that of ResNet-20-4. The single epoch training time on the CIFAR-10 dataset is also nearly four times that of ResNet-20-4, but the increase in training duration on the FashionMNIST dataset is more minor, only about 2.5 times that of ResNet-20-4. This may be mainly because the FashionMNIST dataset itself consists of grayscale images, i.e., the number of channels in the input image is 1, which makes training models easier. On the other hand, CIFAR-10 consists of RGB color images, with the number of channels in the input image being 3, making training more complex than the former. Nonetheless, regardless of the circumstances, the training time of ResNet-20-8 has a significant increase compared to ResNet-20-4. Even so, ResNet-20-8 did not show an improvement on the FashionMNIST dataset compared to ResNet-20-4, but

rather a decline. While the training time significantly increased compared to ResNet-20-4. This indicates that ResNet-20-8 is overwidened for this dataset, and overwidened models cannot provide any performance gain.

Through further analysis, combining time and accuracy, another indicator could be obtained: Marginal gain, which is how much average percentage increase could be acquired in 1 second. Figure 7(a) and Figure 7(b) respectively show the Marginal Gain of each model in the CIFAR-10 and FashionMNIST datasets.



(a) Marginal gains in FashionMNIST for 150 epoch (b) Marginal gains in CIFAR-10 for 300 epoch

Figure 7: Marginal gains of wide and deep model in FashionMNIST and CIFAR-10 datasets.

The results from the above figure draw a common conclusion: wider networks generally outperform deeper ones in terms of overall performance. When comparing models with the same width scaling and depth scaling factors, it can be seen that on the FashionMNIST dataset, the Marginal gain of ResNet-56 is only 0.042, while that of ResNet-20-2 is 0.0695, which is 65.5% higher than the former. The Marginal gain of ResNet-20-4 is 61.0% higher than that of ResNet-110. However, on the CIFAR-10 dataset, the Marginal gain of ResNet-20-2 is 53.6% higher than that of ResNet-56, but the Marginal Gain of ResNet-20-4 is 1.6% lower than that of ResNet-110. Therefore, it can be seen that overall, increasing the width brings more gains on smaller datasets. But for slightly larger

datasets, such as CIFAR-10 used in the experiment, within a certain range, increasing the width can bring more benefits. However, once this range is exceeded, increasing the depth can bring more benefits compared to increasing the width. Based on the results above, when the width coefficient reaches 8 times or more, the benefits brought by the width will be less than the benefits brought by the depth.

In the CIFAR-10 dataset, the highest Marginal gain is achieved by ResNet-20-2. However, their accuracy is not the highest. This is because the complexity of these two models is not as high as other deeper or wider models, and they are sufficient to handle these datasets, hence they have higher Marginal gains. However, it is clear that these indicators cannot effectively measure the accuracy of the model. In order to more comprehensively explore the accuracy of models with different depths and widths, the experiment also included measurements of other performance indicators in the test set, as shown in Table 5.

Model	CIFAR-10 classification accuracy (%)	CIFAR-10 classification F-1 Score	FashionMNIST classification accuracy (%)	FashionMNIST classification F-1 Score
ResNet20	88.90	88.17	93.61	93.60
ResNet56	89.93	89.11	93.51	93.50
ResNet110	89.50	90.58	93.70	93.68
ResNet20-2	90.77	90.63	93.59	94.27
ResNet20-4	91.83	91.32	94.33	94.07
ResNet20-8	93.14	92.88	94.17	94.16

Table 5: The F-1 Score of ResNet with different width and depth in CIFAR-10 and FashionMNIST dataset.

Table 5 shows that the F-1 score remains consistent with the results based on accuracy metrics. For most models, the F1 score is slightly lower than the accuracy, except for ResNet-110 on the CIFAR-10 dataset, where the F-1 score is 1% higher than the accuracy. This discrepancy may be attributed to oscillations during the training of ResNet-110. Nevertheless, regardless of this observation, for relatively smaller datasets (with small image sizes and fewer categories), depth models exhibit robust learning capacity. However, choosing wider and shallower models proves to be the optimal solution.

4.2 GTSRB classification

The comparison results of ResNet of different width and depth in GTSRB dataset are shown in Table 6.

Model	Params	GTSRB classification validation accuracy (%)	GTSRB Single Epoch Training Time (s)
ResNet-24	19.96M	92.01	201
ResNet-50	23.60M	94.44	224
ResNet-101	42.58M	94.84	314.7
ResNet-24-2	79.58M	93.04	438
ResNet-24-4	317.75M	92.55	1255.6

Table 6: The experiment results of ResNet with different width and depth in GTSRB dataset.

The results in Table 6 show that deeper models perform better in GTSRB dataset. ResNet-110 achieves the highest accuracy of 94.84%, which is 2.47% higher than ResNet-24-4. Even ResNet-50 outperforms ResNet-24-4 by 2.04%.

Additionally, Table 7 presents the relevant data for each model. It is evident that models based on the conventional architecture of ResNet have a substantial parameter count. Increasing the width leads to an exponential growth in both the model's training time and parameter count. Conversely, increasing the depth results in a relatively smoother growth in both parameter count and training time for the model. This variation is more visually represented in Figure 8.

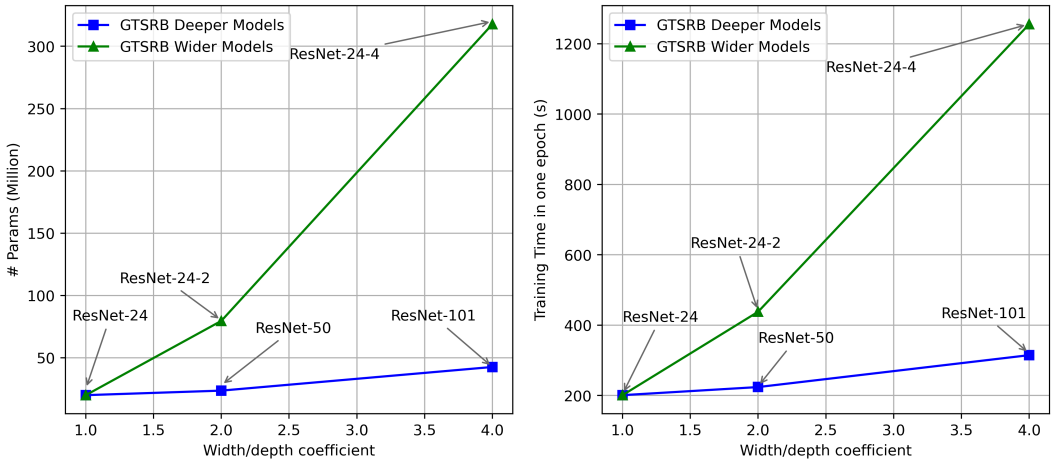


Figure 8: The impact of increasing either width or depth of ResNet on number of parameters and training time in one epoch.

Meanwhile, there is another intriguing phenomenon observed when increasing the width of ResNet-24 to four times its original width. Surprisingly, the accuracy of the model decreases by 0.45% compared to ResNet-24-2. Additionally, the training time of ResNet-24-4 is approximately three times longer than that of ResNet-24-2. It is evident that after doubling the width, the benefits of improving model performance become negative. Therefore, increasing the width is not an optimal choice for enhancing performance on the GTSRB dataset.

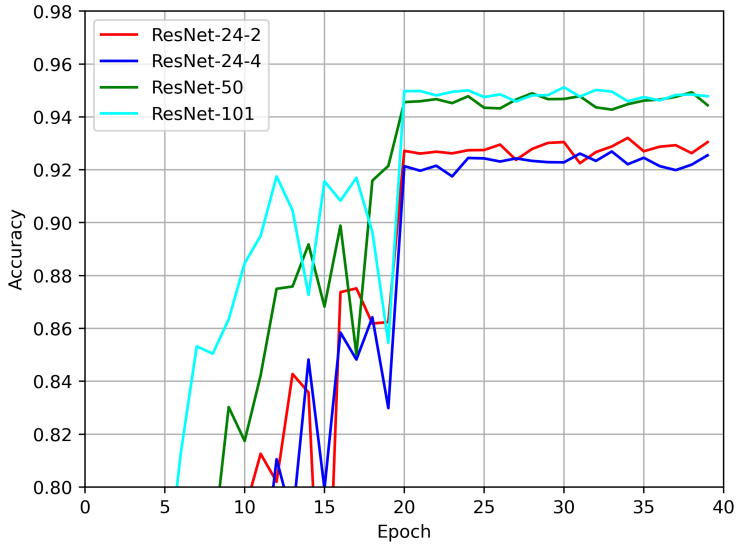


Figure 9: Validation accuracy in GTSRB of ResNet-24-2, ResNet-24-4, ResNet-50, ResNet-101

Figure 9 displays the learning curves of ResNet-24-2, ResNet-24-4, ResNet-50, and ResNet-101, further confirming the earlier conclusion: increasing the depth of the model is the optimal solution for the GTSRB dataset. Moreover, in Figure 9, ResNet-101 demonstrates superior learning capabilities right from the start, surpassing the other three models in accuracy. Although after the first learning rate decay, the accuracy of the model remains within the same range of 94% to 95% as ResNet-50, the overall trend of the learning curve for ResNet-101 is higher than that of ResNet-50.

Table 7 displays the F-1 Score of different models, which align closely with their accuracy metrics, further affirming the aforementioned observations.

Model	GTSRB classification validation accuracy (%)	F-1 Score (%)
ResNet-24	92.01	91.80
ResNet-50	94.44	94.21
ResNet-101	94.84	94.90
ResNet-24-2	93.04	92.46
ResNet-24-4	92.55	92.30

Table 7: The F-1 scores of ResNet-24-2, ResNet-24-4, ResNet-24, ResNet-50 and ResNet-101 in GTSRB dataset.

From Figure 10, it can be clearly seen that for models with the same width and depth coefficients, deep models have a higher marginal gain, which is consistent with the conclusions drawn previously. Through comprehensive analysis, it becomes evident that the model performance on this dataset contrasts sharply with CIFAR-10 and FashionMNIST. This discrepancy stems from the scale difference between the datasets, with GTSRB representing a large-scale dataset while CIFAR-10 and FashionMNIST are small-scale datasets. Therefore, it can be concluded that for ResNet models, increasing depth leads to better performance on large-scale datasets.

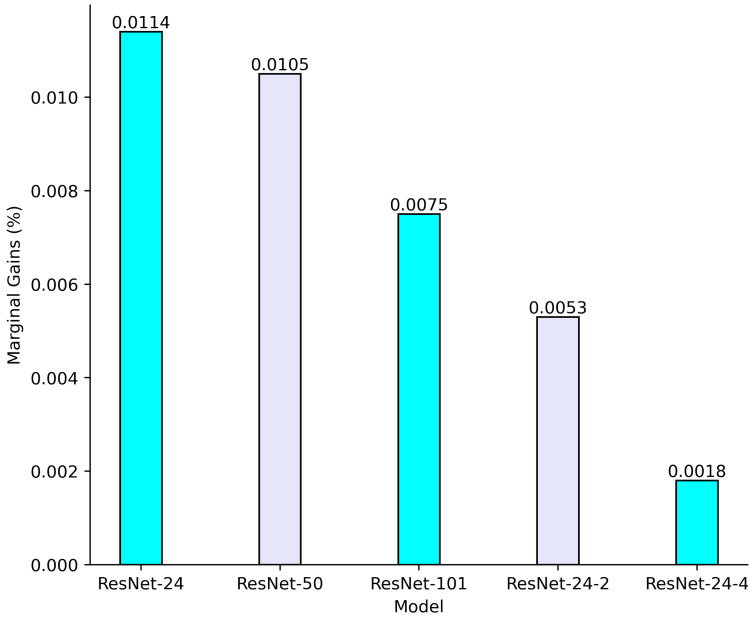


Figure 10: Marginal gains of different width and depth model in GTSRB dataset.

5 Conclusion

This paper, based on the design of residual neural networks, delves into the impact of increasing depth and width on model performance metrics. It explores the question of whether to prioritize depth or width augmentation when dealing with different datasets. Experimental results indicate that for smaller datasets like CIFAR-10 and FashionMNIST, opting for wider and shallower networks yields optimal results. Conversely, for larger datasets such as GTSRB, selecting deeper networks proves to be a superior solution. These findings are expected to provide valuable insights for future research and model training in the field of computer vision.

6 Evaluation and future improvements

The experimental outcomes align with the projected expectations. Hyperparameter tuning has facilitated the attainment of superior training accuracy across diverse datasets. For instance, the discrepancy between the experimental accuracy on the CIFAR-10 dataset and the reported results in the literature is a mere 2%. The juxtaposition of wide and deep models across an array of datasets corroborates the initial hypotheses. Nonetheless, a limitation surfaces in the form of the exclusive use of GTSRB for experiments involving larger datasets, which does not conclusively substantiate the findings. To fortify the conclusions, the inclusion of additional datasets, such as ImageNet or Flowers-101, in the experimental design is recommended. Both aforementioned datasets comprise large-sized images, thereby fulfilling the experimental prerequisites. Moreover, the study confines the evaluation of the performance of wide and deep models to classification tasks. Given that visual tasks extend beyond image recognition to encompass object detection, edge detection, among others, the incorporation of these contexts in future experiments would undoubtedly augment the robustness of the study's conclusions.

Bibliography

Amanatullah. (2023). *Vanishing gradient problem in deep learning: Understanding, intuition, and solutions.* https://medium.com/@amanatulla1606/vanishing-gradient-problem-in-deep-learning-understanding-intuition-and-solutions-da90ef4ecb54.

Han, X., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. https://arxiv.org/pdf/1708.07747v2.pdf.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770-778. doi: 10.1109/cvpr.2016.90

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

Paoletti, M. E., Haut, J. M., Fernandez-Beltran, R., Plaza, J., Plaza, A. J., & Pla, F. (2019). Deep pyramidal residual networks for spectral–spatial hyperspectral image classification. *IEEE Transactions on Geoscience and Remote Sensing*, *57*, 740-754. doi: 10.1109/tgrs.2018.2860125

Ruiz, P. (2018). *Resnets for cifar-10.* https://towardsdatascience.com/resnets-for-cifar-10-e63e900524e0.

Stallkamp, J., Schlipsing, M., Salmen, J., & Igel, C. (2012). Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, *32*, 323-332. doi: 10.1016/j.neunet.2012.02.016

Zagoruyko, S., & Komodakis, N. (2016). Wide residual networks. *Procedings of the British Machine Vision Conference 2016*. doi: 10.5244/c.30.87

Appendix

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.nn.init as init
from torch.autograd import Variable
import torch.optim as optim
from torchvision import datasets
from torchvision import transforms
```

```
device = (torch.device('cuda') if torch.cuda.is_available() else
torch.device('cpu'))
print(f"Training on device {device}.")
```

```
__all__ = ['ResNet', 'resnet20', 'resnet32', 'resnet44', 'resnet56',
'resnet110', 'resnet1202']

def _weights_init(m):
    classname = m.__class__.__name__
    #print(classname)
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        init.kaiming_normal_(m.weight)

class LambdaLayer(nn.Module):
    def __init__(self, lambd):
        super(LambdaLayer, self).__init__()
        self.lambd = lambd

    def forward(self, x):
        return self.lambd(x)


class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1, option='A'):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size=3,
stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=1,
padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != planes:
            if option == 'A':
                self.shortcut = LambdaLayer(lambda x:
                                            F.pad(x[:, :, ::2, ::2], (0, 0,
0, 0, planes//4, planes//4), "constant", 0))
            elif option == 'B':
                self.shortcut = nn.Sequential(
```

```
                    nn.Conv2d(in_planes, self.expansion * planes,
kernel_size=1, stride=stride, bias=False),
                    nn.BatchNorm2d(self.expansion * planes)
                )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out
```

```
class ResNet20_w(nn.Module):
    def __init__(self, block, num_blocks, width_f, num_classes=10):
        super(ResNet20_w, self).__init__()
        widths = [16 * width_f, 32 * width_f, 64 * width_f]
        self.in_planes = widths[0]
        self.conv1 = nn.Conv2d(3, widths[0], kernel_size=3, stride=1,
padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(widths[0])
        self.layer1 = self._make_layer(block, widths[0], num_blocks[0],
stride=1)
        self.layer2 = self._make_layer(block, widths[1], num_blocks[1],
stride=2)
        self.layer3 = self._make_layer(block, widths[2], num_blocks[2],
stride=2)
        self.linear = nn.Linear(widths[2], num_classes)

        self.apply(_weights_init)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion

        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = F.avg_pool2d(out, out.size()[3])
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out
```

```python
def resnet20_2():
    return ResNet20_w(BasicBlock, [3, 3, 3], 2)

def resnet20_4():
    return ResNet20_w(BasicBlock, [3, 3, 3], 4)

def resnet20_8():
    return ResNet20_w(BasicBlock, [3, 3, 3], 8)
```

```python
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_planes = 16

        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.layer1 = self._make_layer(block, 16, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 32, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 64, num_blocks[2], stride=2)
        self.linear = nn.Linear(64, num_classes)

        self.apply(_weights_init)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion

        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = F.avg_pool2d(out, out.size()[3])
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out
```

```python
def resnet20():
    return ResNet(BasicBlock, [3, 3, 3])


def resnet32():
    return ResNet(BasicBlock, [5, 5, 5])


def resnet44():
    return ResNet(BasicBlock, [7, 7, 7])

```

```
def resnet56():
    return ResNet(BasicBlock, [9, 9, 9])


def resnet110():
    return ResNet(BasicBlock, [18, 18, 18])


def resnet1202():
    return ResNet(BasicBlock, [200, 200, 200])
```

```
import pickle

class ModelLogger:
    def __init__(self, path, model_name, params_num):
        self.loss_train = []
        self.acc_train = []
        self.loss_val = []
        self.acc_val = []
        self.params_num = params_num
        self.model_name = model_name
        self.path = path

        basename = os.path.basename(self.path)
        name, ext = os.path.splitext(basename)
        dir_name = os.path.dirname(self.path)
        self.data_path = dir_name + "/" + name + "_data.pkl"

        self.startingTime = ""
        self.start_time = time.time()
        self.firstLog = True
        self.duration = 0.0

    def Log(self, string):
        if self.firstLog == True:
            startingTime = datetime.datetime.now()
            self.firstLog = False
            with open(self.path, 'w') as file:
                file.write("Training {} starting at {}
\n".format(self.model_name, startingTime))
                file.write(f"#Params: {self.params_num}\n")
                file.write("==========================================\n")
                file.write(string + "\n")
        else:
            with open(self.path, 'a') as file:
                file.write(string + "\n")

    def WriteLog(self):
        end_time = time.time()
        with open(self.data_path, 'wb') as file:
            pickle.dump([self.loss_train, self.acc_train, self.loss_val,
self.acc_val, end_time-self.start_time] , file)

    def ReadLog(self):
```

```
        with open(self.data_path, 'rb') as file:
            loaded_array = pickle.load(file)
            self.loss_train, self.acc_train, self.loss_val, self.acc_val,
self.duration = loaded_array
```

```
import datetime
import time
import os

def training_loop(n_epochs, optimizer, scheduler, model, loss_fn,
train_loader, valid_loader, dir, model_name):
    # Write into log
    logger = ModelLogger(dir, model_name, sum([p.numel() for p in
model.parameters()]))

    for epoch in range(1, n_epochs + 1):
        start_time = time.time()

        # Training set
        model.train()
        loss_train, acc_train = train(optimizer, model, loss_fn,
train_loader)
        logger.loss_train.append(loss_train)
        logger.acc_train.append(acc_train)

        # Valid set
        model.eval()
        loss_val, acc_val = valid(model, loss_fn, valid_loader)
        logger.loss_val.append(loss_val)
        logger.acc_val.append(acc_val)

        if not scheduler is None:
            scheduler.step()

        end_time = time.time()
        epoch_time = end_time - start_time

        # log data to file
        logging_str = f"Epoch {epoch}/{n_epochs} \nTraining - Loss:
{loss_train}, Acc: {acc_train} \nValidatation - Loss: {loss_val}, Acc:
{acc_val}\n" + f"Time taken for epoch: {epoch_time:.2f} seconds\n"
        logger.Log(logging_str)

        # Jupyter ouput
        if (epoch == 1 or epoch % 10 == 0):
            print('{} Epoch {}'.format(datetime.datetime.now(), epoch))
            print("Training Loss: {} | Acc: {}".format(loss_train /
len(train_loader), acc_train))
            print("Validation Loss: {} | Acc: {} ".format(loss_val /
len(valid_loader), acc_val))
            print()

    # Write to file
    logger.WriteLog()

```

```
    return logger
```

```
def valid(model, loss_fn, valid_loader):
    loss_valid = 0.0
    correct = 0
    total = 0
    #bar = ProgressBar().start()
    with torch.no_grad():
        for imgs, labels in valid_loader:
            imgs = imgs.to(device=device)
            labels = labels.to(device=device)
            outputs = model(imgs)
            # Calculate the percentage
            _, predicted = torch.max(outputs, dim=1)
            total += labels.shape[0]
            correct += int((predicted == labels).sum())

            loss = loss_fn(outputs, labels)
            loss_valid += loss.item()
        #bar.finish()
    return loss_valid, correct / total

def train(optimizer, model, loss_fn, train_loader):
    loss_train = 0.0
    correct = 0
    total = 0
    #bar = ProgressBar().start()
    for imgs, labels in train_loader:
        imgs = imgs.to(device=device)
        labels = labels.to(device=device)
        outputs = model(imgs)
        # Calculate the percentage
        _, predicted = torch.max(outputs, dim=1)
        total += labels.shape[0]
        correct += int((predicted == labels).sum())

        optimizer.zero_grad()

        loss = loss_fn(outputs, labels)

        # l2_lambda = 0.001
        # l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())
        # loss = loss + l2_lambda * l2_norm

        loss.backward()

        optimizer.step()

        loss_train += loss.item()
    #bar.finish()
    return loss_train, correct / total
```

```
n_classes = 10

```

```
def test(val_loader, model, loss_fn):
    net.eval()
    test_loss = 0
    target_num = torch.zeros((1, n_classes))
    predict_num = torch.zeros((1, n_classes))
    acc_num = torch.zeros((1, n_classes))

    with torch.no_grad():
        for imgs, labels in val_loader:
            imgs, labels = imgs.to(device), labels.to(device)
            outputs = model(imgs)
            loss = loss_fn(outputs, labels)

            test_loss += loss.item()
            _, predicted = outputs.max(1)

            pre_mask = torch.zeros(outputs.size()).scatter_(1,
predicted.cpu().view(-1, 1), 1.)
            predict_num += pre_mask.sum(0)
            tar_mask = torch.zeros(outputs.size()).scatter_(1,
targets.data.cpu().view(-1, 1), 1.)
            target_num += tar_mask.sum(0)
            acc_mask = pre_mask * tar_mask
            acc_num += acc_mask.sum(0)

        recall = acc_num / target_num
        precision = acc_num / predict_num
        F1 = 2 * recall * precision / (recall + precision)

        print('Recall {}, Precision {}, F1-score
{}'.format(torch.sum(recall)/10, torch.sum(precision)/10,
torch.sum(F1)/10))
```

CIFAR-10 Data Processing

```
data_path = './data/CIFAR10/'
cifar10 = datasets.CIFAR10(data_path, train=True, download=True)
cifar10_val = datasets.CIFAR10(data_path, train=False, download=True)
```

```
data_path = "./data/CIFAR10"
cifar10 = datasets.CIFAR10(data_path, train=True, download=False,
transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435,
0.2616)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(30),
    transforms.RandomResizedCrop(32, scale=(0.75, 1.0), ratio=(1.0, 1.0),
antialias=True)
]))

cifar10_val = datasets.CIFAR10(data_path, train=False, download=False,
transform=transforms.Compose([
```

```
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435,
0.2616))
])
```

CIFAR-10 Training

```
train_loader = torch.utils.data.DataLoader(cifar10, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(cifar10_val, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)

model = resnet20_2().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9,
weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[120, 200,
250], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_20_2_cifar = training_loop(300, optimizer, scheduler, model,
loss_fn, train_loader, valid_loader, "./Log/Exp/CIFAR/ResNet20_2",
"Resnet20-2")
```

```
torch.save(model.state_dict(), "./Models/CIFAR/resnet20_2.pth")
```

```
test(valid_loader, ResNet_20_2, loss_fn)
```

```
train_loader = torch.utils.data.DataLoader(cifar10, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(cifar10_val, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)

model = resnet20_4().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9,
weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[120, 200,
250], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_20_4_cifar = training_loop(300, optimizer, scheduler, model,
loss_fn, train_loader, valid_loader, "./Log/Exp/CIFAR/ResNet20_4",
"Resnet20-4")
```

```
torch.save(model.state_dict(), "./Models/CIFAR/resnet20_4.pth")
```

```
test(valid_loader, model, loss_fn)
```

```
train_loader = torch.utils.data.DataLoader(cifar10, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(cifar10_val, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)

model = resnet20_8().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9, weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[120, 200, 250], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_20_8_cifar = training_loop(300, optimizer, scheduler, model, loss_fn, train_loader, valid_loader, "./Log/Exp/CIFAR/ResNet20_8", "Resnet20-8")
```

```
torch.save(model.state_dict(), "./Models/CIFAR/resnet20_8.pth")
```

```
test(valid_loader, model, loss_fn)
```

```
logger_20_2_cifar.ReadLog()
logger_20_4_cifar.ReadLog()
logger_20_8_cifar.ReadLog()
```

```
train_loader = torch.utils.data.DataLoader(cifar10, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(cifar10_val, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)

model = resnet20().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9, weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[120, 200, 250], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_20_cifar = training_loop(300, optimizer, scheduler, model, loss_fn, train_loader, valid_loader, "./Log/Exp/CIFAR/ResNet20", "Resnet20")
```

```
torch.save(model.state_dict(), "./Models/CIFAR/resnet20.pth")
```

```
test(valid_loader, model, loss_fn)
```

```
train_loader = torch.utils.data.DataLoader(cifar10, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(cifar10_val, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)

model = resnet56().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9,
weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[120, 200,
250], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_56_cifar = training_loop(300, optimizer, scheduler, model, loss_fn,
train_loader, valid_loader, "./Log/Exp/CIFAR/ResNet56", "Resnet56")
```

```
torch.save(model.state_dict(), "./Models/CIFAR/resnet56.pth")
```

```
test(valid_loader, model, loss_fn)
```

```
train_loader = torch.utils.data.DataLoader(cifar10, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(cifar10_val, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)

model = resnet110().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9,
weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[120, 200,
250], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_110_cifar = training_loop(300, optimizer, scheduler, model, loss_fn,
train_loader, valid_loader, "./Log/Exp/CIFAR/ResNet110", "Resnet110")
```

```
torch.save(model.state_dict(), "./Models/CIFAR/resnet110.pth")
```

```
test(valid_loader, model, loss_fn)
```

```
logger_20_cifar.ReadLog()
logger_56_cifar.ReadLog()
logger_110_cifar.ReadLog()
```

Plotting graph

```
import matplotlib.pyplot as plt

plt.plot(logger_20_2_cifar.acc_val, label='ResNet-20-2', color='r')
plt.plot(logger_20_4_cifar.acc_val, label='ResNet-20-4', color='b')

```

```
plt.plot(logger_56_cifar.acc_val, label='ResNet-56', ls='--', color='g')
plt.plot(logger_110_cifar.acc_val, label='ResNet-110', ls='--',
color='cyan')

plt.legend()
plt.xlim(100, 300)
plt.ylim(0.85, 0.95)
plt.grid()
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.savefig("./Imgs/CIFAR10_Cmp", dpi=400)
plt.show()
```

```
CIFAR_time_deep = [9.4,14.3,23.6]
CIFAR_Params_deep = [0.27, 0.853, 1.73]
CIFAR_time_wdith = [9.4,9.5,24.8,77.3]
CIFAR_Params_width = [0.27, 1.07, 4.29, 17.1]

CIFAR_Model_names = ["ResNet-20", "ResNet-56", "ResNet-110"]
CIFAR_Model_width_names = ["ResNet-20", "ResNet-20-2", "ResNet-20-4",
"ResNet-20-8"]
```

```
fig, ax = plt.subplots()
plt.plot(CIFAR_Params_deep, CIFAR_time_deep, 's-b', label='CIFAR-10 Deeper
Model')
plt.plot(CIFAR_Params_width, CIFAR_time_wdith, '^-g', label='CIFAR-10 Wider
Model')
plt.legend()
plt.grid()
plt.xlabel("# Params (Million)")
plt.ylabel("Training Time in each epoch")
plt.ylim(0, 80)

ax.annotate('ResNet-20-8', xy=(17, 75), xytext=(14.5, 62),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))
ax.annotate('ResNet-20-4', xy=(4.5, 24), xytext=(7, 32),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))
ax.annotate('ResNet-20-2', xy=(1.1, 8.8), xytext=(3.2, 15),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))
ax.annotate('ResNet-20', xy=(0.27, 9.4), xytext=(0.3, 1.5),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))
ax.annotate('ResNet-56', xy=(0.853, 14.5), xytext=(-0.4, 40),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))
ax.annotate('ResNet-110', xy=(1.73, 24), xytext=(2.5, 45),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))

plt.savefig('CIFAR-10_Wide_Vs_Deep', dpi=400)
plt.show()
```

```
import matplotlib.pyplot as plt

y = [0.0312, 0.0207, 0.0125, 0.0318, 0.0123, 0.00401]
```

```
x_label = ["ResNet-20", "ResNet-56", "ResNet-110", "ResNet-20-2", "ResNet-20-4", "ResNet-20-8"]

bar_width = 0.3

fig = plt.figure(figsize=(7, 6), dpi=100)
ax = plt.axes()
ax.spines["top"].set_visible(False)
ax.spines["right"].set_visible(False)


colors = ['cyan', 'lavender']

bar = plt.bar(x_label, y, width=bar_width, align='center', label='value', color=colors, edgecolor='black')

plt.bar_label(bar, label_type='edge')

plt.xlabel('Model')
plt.ylabel('Marginal Gains (%)')

plt.savefig("Imgs/WidhtAndDepth_CIFAR_Margin.png", bbox_inches='tight', dpi=400)

plt.show()
```

FashionMNIST Data Processing

```
data_path = './data/FashionMinist/'
MNIST = datasets.FashionMNIST(data_path, train=True, download=True)
MNIST_val = datasets.FashionMNIST(data_path, train=False, download=True)
```

```
MNIST = datasets.FashionMNIST(data_path, train=True, download=False, transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(30),
]))

MNIST_val = datasets.FashionMNIST(data_path, train=False, download=False, transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
]))
```

FashionMNIST training

```
train_loader = torch.utils.data.DataLoader(MNIST, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(MNIST_val, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)

model = resnet20_2().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9,
weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[50, 90,
110], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_20_2_mnist = training_loop(150, optimizer, scheduler, model,
loss_fn, train_loader, valid_loader, "./Log/Exp/MNIST/ResNet20_2",
"Resnet20-2")
```

```
torch.save(model.state_dict(), "./Models/MNIST/resnet20_2.pth")
```

```
test(valid_loader, model, loss_fn)
```

```
train_loader = torch.utils.data.DataLoader(cifar10, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(cifar10_val, batch_size=128,
pin_memory=True, num_workers=8, shuffle=True)

model = resnet20_4().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9,
weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[50, 90,
110], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_20_4_mnist = training_loop(150, optimizer, scheduler, model,
loss_fn, train_loader, valid_loader, "./Log/Exp/MNIST/ResNet20_4",
"Resnet20-4")
```

```
torch.save(model.state_dict(), "./Models/MNIST/resnet20_4.pth")
```

```
test(valid_loader, model, loss_fn)
```

```
train_loader = torch.utils.data.DataLoader(cifar10, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(cifar10_val, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)

model = resnet20_8().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9, weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[50, 90, 110], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_20_8_mnist = training_loop(150, optimizer, scheduler, model, loss_fn, train_loader, valid_loader, "./Log/Exp/MNIST/ResNet20_8", "Resnet20-8")
```

```
torch.save(model.state_dict(), "./Models/MNIST/resnet20_8.pth")
```

```
test(valid_loader, model, loss_fn)
```

```
logger_20_2_mnist.ReadLog()
logger_20_4_mnist.ReadLog()
logger_20_8_mnist.ReadLog()
```

```
train_loader = torch.utils.data.DataLoader(MNIST, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(MNIST_val, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)

model = resnet20().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9, weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[50, 90, 110], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_20_mnist = training_loop(150, optimizer, scheduler, model, loss_fn, train_loader, valid_loader, "./Log/Exp/MNIST/ResNet20", "Resnet20")
```

```
torch.save(model.state_dict(), 'ResNet20.pth')
```

```
test(valid_loader, model, loss_fn)
```

```
train_loader = torch.utils.data.DataLoader(MNIST, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(MNIST_val, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)

model = resnet56().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9, weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[50, 90, 110], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_56_mnist = training_loop(150, optimizer, scheduler, model, loss_fn, train_loader, valid_loader, "./Log/Exp/MNIST/ResNet56", "Resnet56")
```

```
torch.save(model.state_dict(), 'ResNet56.pth')
```

```
test(valid_loader, model, loss_fn)
```

```
train_loader = torch.utils.data.DataLoader(MNIST, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)
valid_loader = torch.utils.data.DataLoader(MNIST_val, batch_size=128, pin_memory=True, num_workers=8, shuffle=True)

model = resnet110().to(device=device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9, weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[50, 90, 110], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_110_mnist = training_loop(150, optimizer, scheduler, model, loss_fn, train_loader, valid_loader, "./Log/Exp/MNIST/ResNet110", "Resnet110")
```

```
torch.save(model.state_dict(), 'ResNet110.pth')
```

```
test(valid_loader, model, loss_fn)
```

```
logger_110_mnist.ReadLog()
logger_56_mnist.ReadLog()
logger_20_mnist.ReadLog()
```

Plotting graph

```
import matplotlib.pyplot as plt

plt.plot(logger_20_2.acc_val, label='ResNet-20-2', color='r')
plt.plot(logger_20_4.acc_val, label='ResNet-20-4', color='b')

```

```
plt.plot(logger_56_depth.acc_val, label='ResNet-56', ls='--', color='g')
plt.plot(logger_110_depth.acc_val, label='ResNet-110', ls='--',
color='cyan')

plt.legend()
plt.xlim(0, 150)
plt.ylim(0.9, 0.95)
plt.grid()
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.savefig("./Imgs/MNIST_Cmp", dpi=400)
plt.show()
```

```
MNIST_time_deep = [8.1,14.8,25.3]
MNIST_time_width = [8.1,9.1,15.48,39.4]
```

```
fig, ax = plt.subplots()
plt.plot(CIFAR_Params_deep, MNIST_time_deep, 's-b', label='FashionMNIST
Deeper Model')
plt.plot(CIFAR_Params_width, MNIST_time_width, '^-g', label='FashionMNIST
Wider Model')

ax.annotate('ResNet-20-8', xy=(17, 39), xytext=(14, 32),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))
ax.annotate('ResNet-20-4', xy=(4.5, 15), xytext=(4, 8),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))
ax.annotate('ResNet-20-2', xy=(1.1, 8.8), xytext=(2, 4),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))
ax.annotate('ResNet-20', xy=(0.27, 7.8), xytext=(-0.4, 1.5),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))
ax.annotate('ResNet-56', xy=(0.853, 15), xytext=(0, 28),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))
ax.annotate('ResNet-110', xy=(1.8, 25.5), xytext=(3, 26),
arrowprops=dict(arrowstyle = '->', color = 'dimgray'))

plt.legend()
plt.grid()

plt.xlabel("# Params (Million)")
plt.ylabel("Training Time in each epoch")
plt.ylim(0, 40)
plt.savefig('FashioMNIST_Wide_Vs_Deep', dpi=400)
plt.show()
```

```
y = [0.0771,0.0420,0.0251,0.0695,0.0404,0.0160]
x_label = ["ResNet-20", "ResNet-56", "ResNet-110", "ResNet-20-2", "ResNet-
20-4", "ResNet-20-8"]

bar_width = 0.3

fig = plt.figure(figsize=(7, 6), dpi=100)
ax = plt.axes()
ax.spines["top"].set_visible(False)
```

```
ax.spines["right"].set_visible(False)


colors = ['cyan', 'lavender']

bar = plt.bar(x_label, y, width=bar_width, align='center', label='value',
color=colors, edgecolor='black')

plt.bar_label(bar, label_type='edge')

plt.xlabel('Model')
plt.ylabel('Marginal Gains (%)')

plt.savefig("Imgs/WidhtAndDepth_MNIST_Margin.png", bbox_inches='tight',
dpi=400)

plt.show()
```

GTSRB Data Processing

```
import zipfile

with zipfile.ZipFile('./data/GTSRB/archive_2.zip', 'r') as zipf:
    zipf.extractall('./data/GTSRB')
```

```
data_path = "./data/GTSRB/"
```

```
meta_df = pd.read_csv('./data/GTSRB/Meta.csv')
train_df = pd.read_csv(data_path + 'Train.csv')
test_df = pd.read_csv(data_path + 'Test.csv')
```

```
train_data_path = os.path.join(data_path, 'Train')
test_data_path = os.path.join(data_path, 'Test')
meta_data_path = os.path.join(data_path, 'Meta')
```

```
transforms_train = transforms.Compose([transforms.RandomRotation(45),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.5, 0.5, 0.5],
[0.5, 0.5, 0.5]),
                                       transforms.Resize((225, 225),
antialias=True),
                                       transforms.RandomHorizontalFlip(p=0.5),
                                       transforms.RandomVerticalFlip(p=0.5)])

transforms_val = transforms.Compose([transforms.ToTensor(),
                                         transforms.Normalize([0.5, 0.5, 0.5],
[0.5, 0.5, 0.5]),
                                         transforms.Resize((225, 225),
antialias=True)])
```

```
class TSignsDataset(Dataset):
```

```
    def __init__(self, df, root_dir,transform=None):
        self.df = df
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.df)

    def __getitem__(self,index):
        image_path = os.path.join(self.root_dir,self.df.iloc[index,7]) #the column of paths in df is 7
        image = Image.open(image_path)
        y_class = torch.tensor(self.df.iloc[index, 6]) #the column of ClsassId in df is 6

        if self.transform:
            image = self.transform(image)

        return (image, y_class)
```

```
train_set = TSignsDataset(train_df,data_path,transform=transforms_train)
val_set = TSignsDataset(test_df,data_path,transform=transforms_val)
```

```
train_loader = DataLoader(dataset=train_set, batch_size=64, shuffle=True, pin_memory=True)
val_loader = DataLoader(dataset=val_set, batch_size=64, shuffle=False, pin_memory=True)
```

ResNet Model Construction

```
class block(nn.Module):
    def __init__(
        self, in_channels, out_channels, identity_downsample=None, stride=1):
        super().__init__()
        self.expansion = 4
        self.conv1 = nn.Conv2d(in_channels,out_channels,kernel_size=1,stride=1,padding=0,bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels,out_channels,kernel_size=3,stride=stride,padding=1,bias=False,)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.conv3 = nn.Conv2d(out_channels,out_channels * self.expansion,kernel_size=1,stride=1,padding=0,bias=False,)
        self.bn3 = nn.BatchNorm2d(out_channels * self.expansion)
        self.relu = nn.ReLU()
        self.identity_downsample = identity_downsample
        self.stride = stride

    def forward(self, x):
        identity = x.clone()
```

```
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.conv3(x)
        x = self.bn3(x)

        if self.identity_downsample is not None:
            identity = self.identity_downsample(identity)

        x += identity
        x = self.relu(x)
        return x
```

```
class ResNet(nn.Module):
    def __init__(self, block, layers, image_channels, num_classes, width_f):
        super(ResNet, self).__init__()
        self.in_channels = 64 * width_f
        widths = [64 * width_f, 128 * width_f, 256 * width_f, 512 * width_f]
        self.conv1 = nn.Conv2d(image_channels, widths[0], kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(widths[0])
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # Essentially the entire ResNet architecture are in these 4 lines below
        self.layer1 = self._make_layer(block, layers[0], out_channels=widths[0], stride=1)
        self.layer2 = self._make_layer(block, layers[1], out_channels=widths[1], stride=2)
        self.layer3 = self._make_layer(block, layers[2], out_channels=widths[2], stride=2)
        self.layer4 = self._make_layer(block, layers[3], out_channels=widths[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(widths[3] * 4, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
```

```
        x = x.reshape(x.shape[0], -1)
        x = self.fc(x)

        return x

    def _make_layer(self, block, num_residual_blocks, out_channels, stride):
        identity_downsample = None
        layers = []

        if stride != 1 or self.in_channels != out_channels * 4:
            identity_downsample = nn.Sequential(nn.Conv2d(self.in_channels,out_channels * 4,kernel_size=1,stride=stride,bias=False)

,nn.BatchNorm2d(out_channels * 4))

        layers.append(block(self.in_channels, out_channels, identity_downsample, stride))

        self.in_channels = out_channels * 4

        for i in range(num_residual_blocks - 1):
            layers.append(block(self.in_channels, out_channels))

        return nn.Sequential(*layers)
```

```
def ResNet24_2(img_channel=3, num_classes=1000):
    return ResNet(block, [3, 3, 3, 3], img_channel, num_classes, 2)

def ResNet24_4(img_channel=3, num_classes=1000):
    return ResNet(block, [3, 3, 3, 3], img_channel, num_classes, 4)
```

```
def ResNet24(img_channel=3, num_classes=1000):
    return ResNet(block, [3, 3, 3, 3], img_channel, num_classes, 1)

def ResNet50(img_channel=3, num_classes=1000):
    return ResNet(block, [3, 4, 6, 3], img_channel, num_classes, 1)

def ResNet101(img_channel=3, num_classes=1000):
    return ResNet(block, [3, 4, 23, 3], img_channel, num_classes, 1)
```

GTSRB Training

```
model = ResNet24_2(img_channel=3, num_classes=43)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9, weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[20], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_24_2 = training_loop(40, optimizer, scheduler, model, loss_fn,
train_loader, val_loader, "./Log/GTSRB/ResNet_24_2", "Resnet24_2")
```

```
torch.save(model.state_dict(), "./Models/GTSRB/resnet24_2.pth")
```

```
test(val_loader, model, loss_fn)
```

```
model = ResNet24_4(img_channel=3, num_classes=43).to(device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9,
weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[20],
gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_24_4 = training_loop(40, optimizer, scheduler, model, loss_fn,
train_loader, val_loader, "./Log/GTSRB/ResNet_24_4", "Resnet24_4")
```

```
torch.save(model.state_dict(), "./Models/GTSRB/resnet24_4.pth")
```

```
test(val_loader, model, loss_fn)
```

```
model = ResNet24(img_channel=3, num_classes=43).to(device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9,
weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[20],
gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_24 = training_loop(40, optimizer, scheduler, model, loss_fn,
train_loader, val_loader, "./Log/GTSRB/ResNet_24", "Resnet24")
```

```
torch.save(model.state_dict(), "./Models/GTSRB/resnet24.pth")
```

```
test(val_loader, model, loss_fn)
```

```
model = ResNet50(img_channel=3, num_classes=43).to(device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9,
weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[20],
gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_24 = training_loop(40, optimizer, scheduler, model, loss_fn,
train_loader, val_loader, "./Log/GTSRB/ResNet_50", "Resnet50")
```

```
torch.save(model.state_dict(), "./Models/GTSRB/resnet50.pth")
```

```
test(val_loader, model, loss_fn)
```

```
model = ResNet101(img_channel=3, num_classes=43).to(device)
optimizer = optim.SGD(model.parameters(), lr=1e-1, momentum=0.9, weight_decay=1e-4, nesterov=True)
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[20], gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
```

```
logger_101 = training_loop(40, optimizer, scheduler, model, loss_fn, train_loader, val_loader, "./Log/GTSRB/ResNet_101", "Resnet101")
```

```
torch.save(model.state_dict(), "./Models/GTSRB/resnet101.pth")
```

```
test(val_loader, model, loss_fn)
```

Plotting

```
import matplotlib.pyplot as plt

plt.plot(logger_24_2.acc_val, label='ResNet-24-2', color='r')
plt.plot(logger_24_4.acc_val, label='ResNet-24-4', color='b')

plt.plot(logger_50.acc_val, label='ResNet-50', ls='-', color='g')
plt.plot(logger_101.acc_val, label='ResNet-101', ls='-', color='cyan')

plt.legend()
plt.xlim(0, 40)
plt.ylim(0.8, 0.98)
plt.grid()
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.savefig("./GTSRB_Cmp", dpi=400)
plt.show()
```

```
GTSRB_Model_depth_names = ["ResNet-20", "ResNet-56", "ResNet-110"]
GTSRB_Model_width_names = ["ResNet-20", "ResNet-20-2", "ResNet-20-4"]

GTSRB_time_depth = [201,224,314.7]
GTSRB_Params_depth = [19.96, 23.60, 42.58]

GTSRB_time_width = [201, 438, 1255.6]
GTSRB_Params_width = [19.96, 79.58, 317.75]
```

```
factors = [1, 2, 4]
plt.figure(figsize=(12, 5))

plt.subplot(121)
plt.plot(factors, GTSRB_Params_depth, 's-b', label='GTSRB Deeper Models')
plt.plot(factors, GTSRB_Params_width, '^-g', label='GTSRB Wider Models')
```

```
plt.legend()
plt.grid()
plt.xlabel("Width/depth coefficient")
plt.ylabel("# Params (Million)")
#plt.ylim(0, 80)

plt.subplot(122)
plt.plot(factors, GTSRB_time_depth, 's-b', label='GTSRB Deeper Models')
plt.plot(factors, GTSRB_time_wdith, '^-g', label='GTSRB Wider Models')
plt.legend()
plt.grid()
plt.xlabel("Width/depth coefficient")
plt.ylabel("Training Time in one epoch (s)")

plt.savefig('GTSRB_Wide_Vs_Deep_params_time', dpi=400)
plt.show()
```