Machine Learning Course Project

0 Abstract

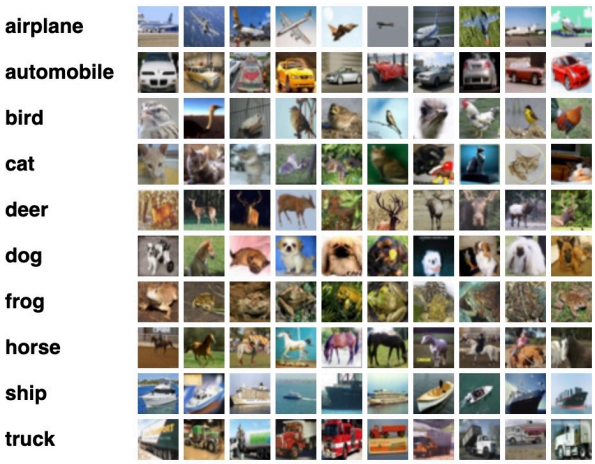
In this project, I address the problem of image classification on the CIFAR-10 dataset with a series of CNN-based neural networks. The Machine Learning lectures have provided me with a deeper insight into the ML models and principles behind neural networks. Traditionally, many computer vision tasks are surrounded by CNN architectures, as the basis of most of the problems is to classify an image into known labels. Some famous algorithms for object detection like SSD(single shot multi-box detection) and YOLO (You Only Look

Once) are also built on CNN. I think it is proper for me to start my first machine learning repository with CIFAR-10, which is vanilla while still a little bit challenging. To this end, I experiment with various models, including VGG nets, mobile nets, etc. To evaluate the models' robustness and generalization, I test trained models on test sets with different noises, which should test models comprehensively. I also do systematic ablation studies on both the network architecture and some hyperparameters. Here I demonstrate that the results of the models are very satisfactory. The official code of this project has been released on my personal GitHub repository ToyCIFAR10.

1 Introduction

Image Classification is a fundamental task that attempts to comprehend an entire image as a whole. The goal is to classify the image by assigning it to a specific label. Typically, Image Classification refers to images in which only one object appears and is analyzed. In contrast, object detection involves both classification and localization tasks and is used to analyze more realistic cases in which multiple objects may exist in an image.

The CIFAR-10 dataset (Canadian Institute for Advanced Research, 10 classes) is a subset of the Tiny Images dataset and consists of 60000 32x32 color images. The images are labeled with one of 10 mutually exclusive classes: airplane, automobile (but not truck or pickup truck), bird, cat, deer, dog, frog, horse, ship, and truck (but not pickup truck). There are 6000 images per class with 5000 training and 1000 testing images per class.



I leverage various convolutional Neural Networks and several famous architectures based on CNN in this project, which are simple but effective. I mainly choose VGG nets (from VGG-11 to VGG-19) and MobileNet considering their relatively low demands on GPU. The components and structure of the models are shaken to figure out the exact role each part of the model plays.

2 Methods

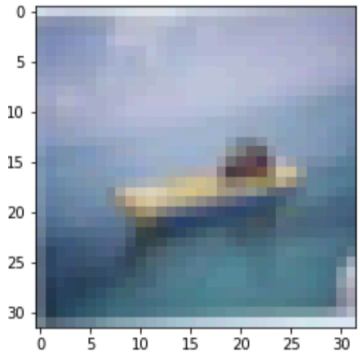
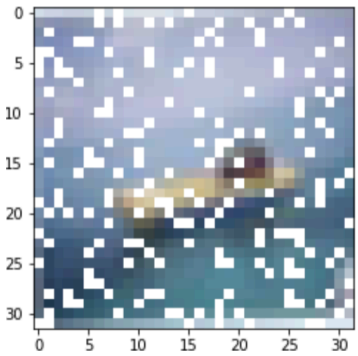
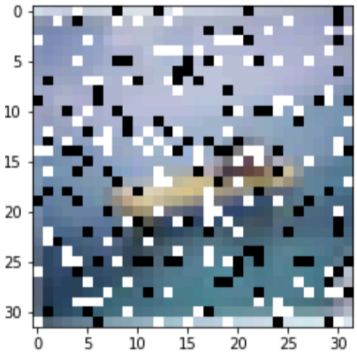
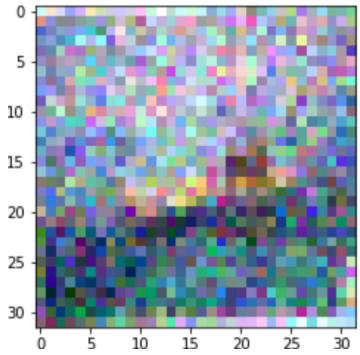
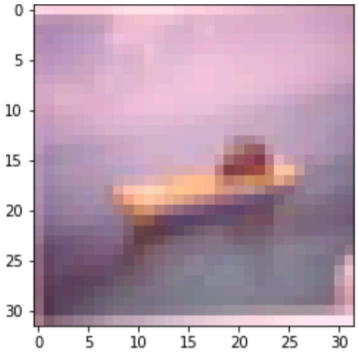
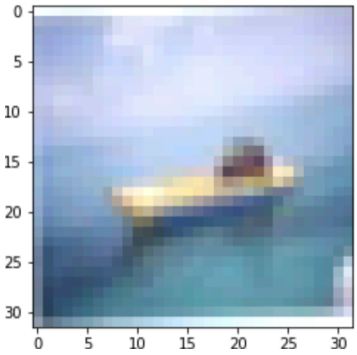
2.1 Dataset

2.1.1 Trainset

The Torchvision module provides off-the-shelf access to the CIFAR-10 dataset, which is quite convenient. However, if the training data is directly fed into the model, the results are far from satisfying. So we take several methods to preprocess the training data, which can be called data augmentation. By applying small disturb or changes to the training data, Data augmentation can not only increase the number of training

data but also make our model more robust to noises. We combine many basic but effective operations together to augment our training data. The basic operations include:

- Random noise: Randomly change the values of pixels in an image.
- Gauss noise: The values that the noise can take on are Gaussian-distributed.
- Salt-and-pepper noise: White and black pixels sparsely occur in the image.
- Flip: Horizontally vertically or flip the given image randomly with a given probability
- Normalize: Normalize the images in separate RGB channels
- Whitening: Convert the pixel values to [0, 1]
- Brightness and Contrasts: change the images' brightness and contrasts

Original	Random noise	Salt-and-Pepper noise
		
Gauss noise	**Brightness**	**Contrast**
		

To customize different noises types and manipulation of the dataset, we implement our own dataset class: `class ToyCIFAR10(torch.utils.dataset)`, instead of using the TorchVision API. In this class, we first store the train set and test set separately, and the labels and images are stored in different `.pt` files. Then we judge the object as the train or test dataset, and we do different data preprocessing accordingly. We implement our noise functions in the `dataset.py/class Noise`.

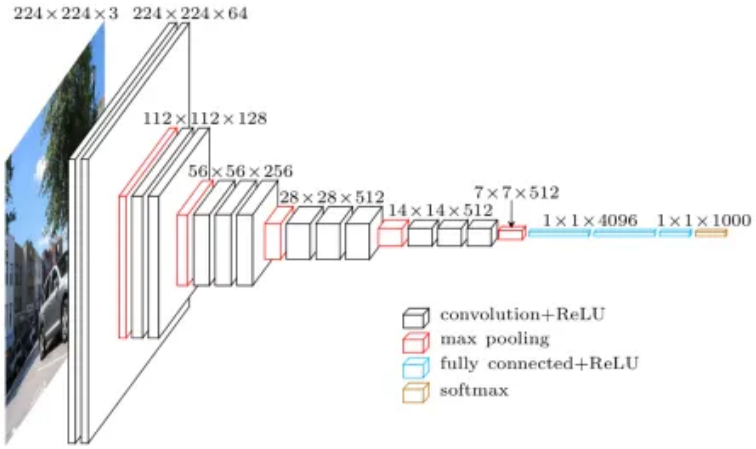
2.1.2 Testset

To test our models' robustness, we evaluate experiments on the original test set, as well as on the test set with noises. The noises include changing the brightness, adding random noises, and so on. We also find that after data augmentation, the accuracy on both original and noised test sets improves a lot.

2.2 Network

2.2.1 VGG net

VGG stands for Visual Geometry Group; it is a standard deep Convolutional Neural Network (CNN) architecture with multiple layers. The “deep” refers to the number of layers with VGG-16 or VGG-19 consisting of 16 and 19 convolutional layers.



Let’s take a brief look at the architecture of VGG:

- **Convolutional Layers**: VGG’s convolutional layers leverage a minimal receptive field, i.e., 3×3, the smallest possible size that still captures up/down and left/right. Moreover, there are also 1×1 convolution filters acting as a linear transformation of the input. This is followed by a ReLU unit, which is a huge innovation from AlexNet that reduces training time. ReLU stands for rectified linear unit activation function; it is a piecewise linear function that will output the input if positive; otherwise, the output is zero. The convolution stride is fixed at 1 pixel to keep the spatial resolution preserved after convolution (stride is the number of pixel shifts over the input matrix).
- **Hidden Layers:** All the hidden layers in the VGG network use ReLU.
- **Fully-Connected Layers:** The VGGNet has three fully connected layers. Out of the three layers, the first two have 4096 channels each, and the third has 1000 channels, 1 for each class.

VGG brought with it a massive improvement in accuracy and a speed improvement as well. This was primarily because of improving the depth of the model and also introducing pre-trained models. The increase in the number of layers with smaller kernels saw an increase in non-linearity which is always a positive in deep learning. Besides, VGG brought with it various architectures built on a similar concept.

One major disadvantage that I found was that this model experiences the vanishing gradient problem, which prevents the classification accuracy from improving. This wasn’t the case with any of the other models. The vanishing gradient problem was solved with the ResNet architecture.

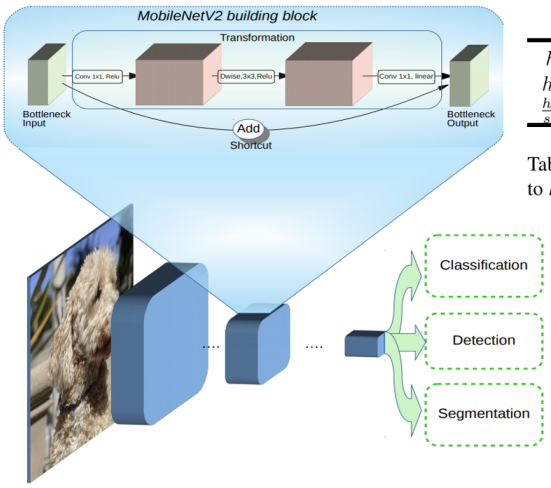
The configuration file for our implementation of VGG models is shown below, where 'M' stands for MaxPooling layer, and digits mean convolutional layers that have a corresponding number of input channels followed by a ReLU activation layer.

```
cfg = {
    'vgg11': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'vgg13': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'vgg16': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512,
512, 512, 'M'],
    'vgg19': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512,
'M', 512, 512, 512, 512, 'M'],
}
```

For the batch norm models, we attach a BatchNorm layer after each Convolutional layer and before the ReLU.

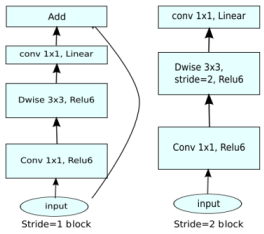
The layers are arranged sequentially. At the end of each model, there is a simple MLP to classify the feature maps to the final 10 classes.

2.2.2 Mobile net



Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Table 1: *Bottleneck residual block* transforming from k to k' channels, with stride s, and expansion factor t.



(d) Mobilenet V2

MobileNets are based on a streamlined architecture that uses depthwise separable convolutions to build lightweight deep neural networks. In many real-world applications such as robotics, self-driving car, and augmented reality, the recognition tasks need to be carried out in a timely fashion on a computationally limited platform. This paper describes an efficient network architecture and a set of two hyper-parameters to build very small, low latency models that can be easily matched to the design requirements for mobile and embedded vision applications. MobileNet released by Google is majorly used for mobile applications. The following figure shows the comparison between different networks and how MobileNet outperforms it using depth-wise separable convolutions.

2.3 Hyperparameters

2.3.1 Kernel Size and Padding

We are using the most popular choice that is used by every Deep Learning practitioner out there, and that is 3x3 kernel size. One of the reasons to prefer small kernel sizes over the fully connected network is that it reduces computational costs and weight sharing that ultimately leading to lesser weights for back-propagation. So then came VGG net in 2015 which replaced such large convolution layers with 3x3 convolution layers but with a lot of filters. And since then, 3x3 sized kernel habecomeme as a popular

choice.

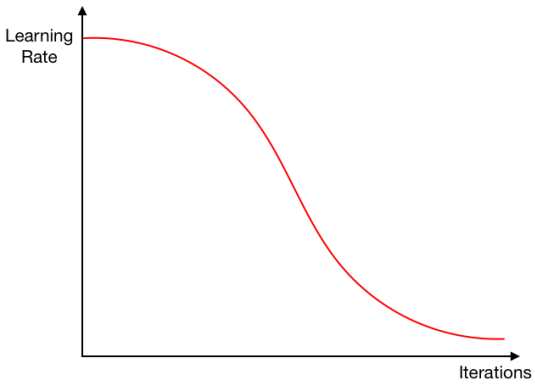
Convolution operation shrinks the size of the image. But shrinking the image size is an issue if we want to stack many layers (size will reduce to 1x1 quickly). Padding input image with zeros allows them to have the same size for the input and the output.

2.3.2 Stride

The amount of movement between applications of the filter to the input image is referred to as the stride, and it is almost always symmetrical in height and width dimensions. Stride convolution can be useful for large images. Downsampling may be desirable in some cases where a deeper knowledge of the filters used in the model or of the model architecture allows for some compression in the resulting feature maps.

2.3.3 Learning rate

We set the initial learning rate to be 0.05 for all models, and we moderate it with an off-the-shelf cosine annealing strategy in PyTorch: `torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=args.epoch)`. The learning rate sways periodically like a cosine curve. In our implementation, the period is set as the max epoch number, so the learning rate will drop like the curve below.



2.3.4 Other details

We choose cross-entropy loss as our criterion for computing loss function. The classic SGD optimizer is used and the parameters are `momentum=0.9, weight_decay=5e-4`, and they prove effective.

2.4 Evaluation

We evaluate our models based on many benchmarks. The original test set and noised test set are evaluated to judge the models' robustness. Furthermore, we test on all 10 classes, detailed accuracy rates are provided for each class. During training, the loss in training and accuracy rate on the test set are printed. Since we also record the accuracy after each epoch, we can visualize the accuracy growth in a table, which is quite clear and straight. This part is implemented in `utils/eval_on_test_set` function.

We save the model after they have been trained for every 100 epochs. Also, the model that performs best and the latest model are both saved. We can resume the training where they are suspended last time.

3 Experiments

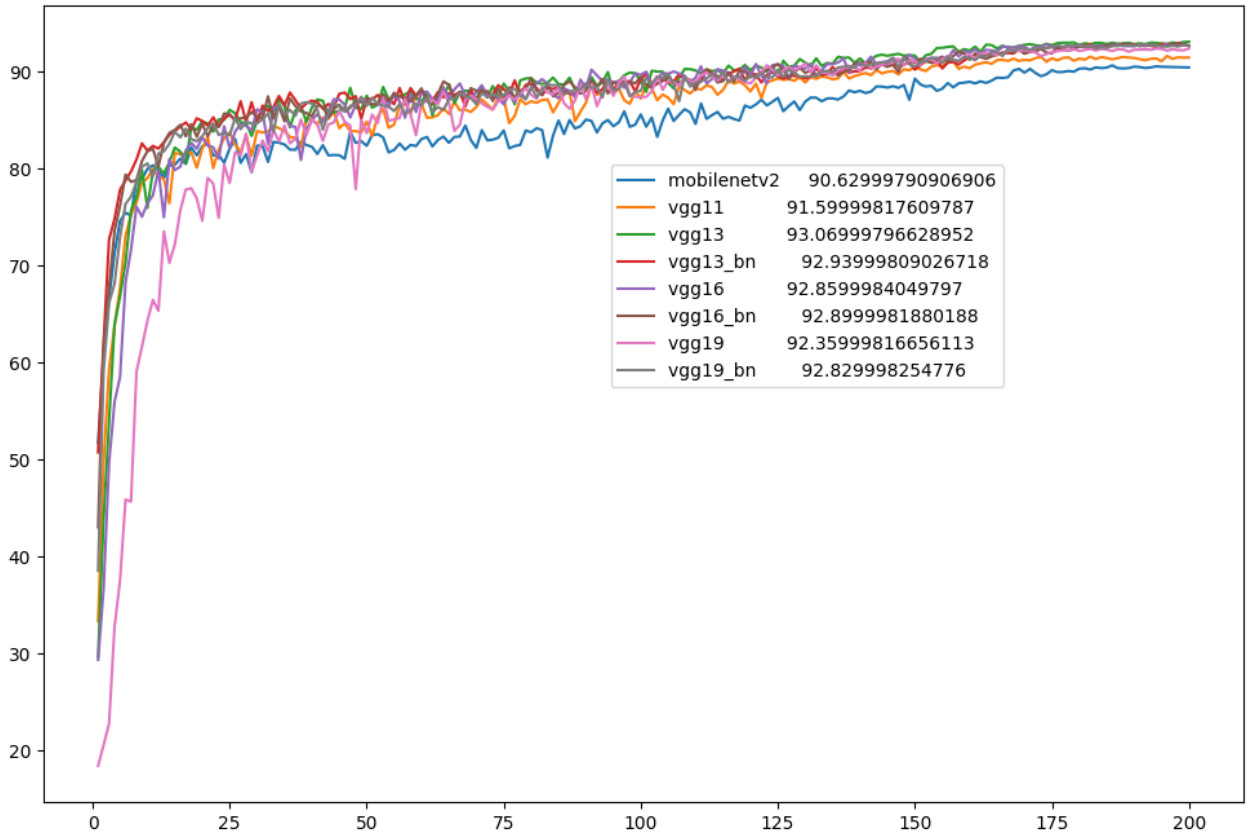
3.1 Implementation Details

We implement our framework in PyTorch. Both networks are trained withSGD optimizer with an initial learning rate of 0.05. We train each model for 200 epochs. In each epoch, we shuffle the images and feed them into the network with a batch size of 128 images. We train the networks with a single NVIDIA GeForce RTX 2080 Ti.

	VGG-11	VGG-13	VGG-16	VGG-19	MobileNetv2
Number of parameters (million)	9.75	9.94	15.25	20.55	2.3
Time per epoch (min)	0.87	1.00	1.07	1.10	0.98
Memory Usage (roughly) (MiB)	1903	2103	2177	2217	4955

3.2 Results on the original test set

We first train our models with no noise images and only basic processes like normalization and `ToTensor`. Then we test them on the original test dataset. The best results and the models' accuracy growth curve are listed below.



We also record the models' accuracy rate change on the test set with an increasing interval 20 of training epochs in a table.

Accuracy rate(%) num of epochs	20	40	60	80	100	120	140	160	180	200	*Best*
Mobilenetv2	82.28	82.24	83.04	83.74	85.57	86.26	88.01	88.94	90.29	90.42	90.63
VGG-11	82.03	84.96	86.65	86.93	87.21	87.94	89.69	90.33	91.33	91.44	91.60
VGG-13	84.86	86.28	86.88	89.33	88.97	90.12	91.35	92.15	92.72	93.07	93.19
VGG-16	83.22	85.00	87.87	88.22	89.65	89.55	90.62	91.92	92.76	92.64	92.86
VGG-19	74.61	85.07	86.77	87.60	87.37	88.77	90.10	91.71	92.13	92.36	92.36
VGG-13 (Batch norm)	84.89	86.99	87.95	88.74	89.21	89.86	90.06	91.50	92.65	92.63	92.94
VGG-16 (Batch norm)	84.61	85.35	88.26	89.31	89.46	89.39	90.93	91.21	92.58	92.70	92.90
VGG-19 (Batch norm)	83.33	86.69	86.49	88.42	88.02	89.56	90.97	91.42	92.68	92.65	92.83

All the models converge quickly, but it takes the majority of training time to push the accuracy rate from 80% to 90% or higher. So the learning rate is set to drop quickly using the cosine annealing method. When the training error drops to lower than 1 percent, the accuracy on the test set shakes more frequently, which means that the model is overfitted. For more training details, please refer to the log files in my repository.

3.3 Results on the noisy test set

We also test our models' robustness on the test set with noise, in which we add diverse noises to the images.

Note that we do not use noise images in training in this part.

For brevity, we only test our VGG-13 (Batch norm) model with all 5 kinds of noises.

Random noise

A random number of pixels are set to 1(255) in random noise mode. We add the number of noise pixels gradually, noting that the total number of pixels in an image is 32 * 32 = 1024. We list the accuracy of each class as well.

Num of noise pixels	**20**	**50**
VGG-13 (Batch norm)	load model: outputs/vgg13_bn/default/best.pth accuracy on test set = 82.51999777555466 percent Test Accuracy of airplane: 86% (864/1000) Test Accuracy of automobile: 89% (898/1000) Test Accuracy of Bird: 70% (702/1000) Test Accuracy of Cat: 73% (737/1000) Test Accuracy of Deer: 80% (804/1000) Test Accuracy of Dog: 75% (751/1000) Test Accuracy of Frog: 84% (842/1000) Test Accuracy of Horse: 89% (899/1000) Test Accuracy of Ship: 83% (831/1000) Test Accuracy of Truck: 92% (924/1000)	load model: outputs/vgg13_bn/default/best.pth accuracy on test set = 72.79999816417696 percent Test Accuracy of airplane: 80% (806/1000) Test Accuracy of automobile: 81% (810/1000) Test Accuracy of Bird: 59% (590/1000) Test Accuracy of Cat: 62% (622/1000) Test Accuracy of Deer: 68% (682/1000) Test Accuracy of Dog: 63% (639/1000) Test Accuracy of Frog: 74% (741/1000) Test Accuracy of Horse: 81% (812/1000) Test Accuracy of Ship: 72% (725/1000) Test Accuracy of Truck: 85% (853/1000)
	100	**200**
	load model: outputs/vgg13_bn/default/best.pth accuracy on test set = 62.719998955726616 percent Test Accuracy of airplane: 72% (728/1000) Test Accuracy of automobile: 67% (678/1000) Test Accuracy of Bird: 49% (496/1000) Test Accuracy of Cat: 53% (537/1000) Test Accuracy of Deer: 56% (564/1000) Test Accuracy of Dog: 53% (537/1000) Test Accuracy of Frog: 63% (639/1000) Test Accuracy of Horse: 70% (703/1000) Test Accuracy of Ship: 61% (612/1000) Test Accuracy of Truck: 77% (778/1000)	load model: outputs/vgg13_bn/default/best.pth accuracy on test set = 48.959999322891235 percent Test Accuracy of airplane: 66% (660/1000) Test Accuracy of automobile: 44% (442/1000) Test Accuracy of Bird: 35% (357/1000) Test Accuracy of Cat: 43% (437/1000) Test Accuracy of Deer: 43% (430/1000) Test Accuracy of Dog: 39% (393/1000) Test Accuracy of Frog: 53% (530/1000) Test Accuracy of Horse: 56% (561/1000) Test Accuracy of Ship: 45% (452/1000) Test Accuracy of Truck: 63% (634/1000)

Salt-and-Pepper noise

Salt-and-pepper noise means that some pixels are set to 1(255) while some are set to 0. We alter the probability of noise pixels to test our VGG-13 (Batch norm) model. Especially, 1024 * prob pixels are set to white pixels, and 1024 * prob pixels are set to black.

Prob of noise pixels	**0.01**	**0.025**
VGG-13 (Batch norm)	load model: outputs/vgg13_bn/default/best.pth accuracy on test set = 80.9899982213974 percent Test Accuracy of airplane: 85% (856/1000) Test Accuracy of automobile: 88% (889/1000) Test Accuracy of Bird: 67% (676/1000) Test Accuracy of Cat: 69% (694/1000) Test Accuracy of Deer: 79% (797/1000) Test Accuracy of Dog: 73% (733/1000) Test Accuracy of Frog: 85% (852/1000) Test Accuracy of Horse: 87% (879/1000) Test Accuracy of Ship: 81% (818/1000) Test Accuracy of Truck: 90% (905/1000)	load model: outputs/vgg13_bn/default/best.pth accuracy on test set = 70.1799984574318 percent Test Accuracy of airplane: 79% (790/1000) Test Accuracy of automobile: 77% (779/1000) Test Accuracy of Bird: 53% (538/1000) Test Accuracy of Cat: 57% (573/1000) Test Accuracy of Deer: 68% (683/1000) Test Accuracy of Dog: 56% (569/1000) Test Accuracy of Frog: 72% (727/1000) Test Accuracy of Horse: 80% (800/1000) Test Accuracy of Ship: 72% (724/1000) Test Accuracy of Truck: 83% (835/1000)
	0.05	**0.1**
	load model: outputs/vgg13_bn/default/best.pth accuracy on test set = 57.49999928474426 percent Test Accuracy of airplane: 69% (691/1000) Test Accuracy of automobile: 62% (628/1000) Test Accuracy of Bird: 37% (374/1000) Test Accuracy of Cat: 50% (506/1000) Test Accuracy of Deer: 50% (508/1000) Test Accuracy of Dog: 42% (423/1000) Test Accuracy of Frog: 61% (610/1000) Test Accuracy of Horse: 66% (669/1000) Test Accuracy of Ship: 63% (637/1000) Test Accuracy of Truck: 70% (704/1000)	load model: outputs/vgg13_bn/default/best.pth accuracy on test set = 42.2299987077713 percent Test Accuracy of airplane: 54% (548/1000) Test Accuracy of automobile: 38% (385/1000) Test Accuracy of Bird: 25% (256/1000) Test Accuracy of Cat: 36% (366/1000) Test Accuracy of Deer: 35% (354/1000) Test Accuracy of Dog: 32% (322/1000) Test Accuracy of Frog: 49% (494/1000) Test Accuracy of Horse: 48% (480/1000) Test Accuracy of Ship: 45% (456/1000) Test Accuracy of Truck: 56% (562/1000)

We can see that some classes (bird, cat, dog, deer) are more vulnerable to random noises. Actually, they are also vulnerable to other types of noises as well, which can be attributed to their pixel structures and distribution.

Gauss noise

Gaussian noise is statistical noise having a probability density function (PDF) equal to that of the normal distribution. In other words, the values that the noise can take on are Gaussian-distributed. Here we increase the variance of the distribution to increase the noise.

Variance of Gauss distribution / accuracy rate(%)	**0.001**	**0.005**
VGG-13 (Batch norm)	86.44	72.61
	0.01	**0.02**
	61.76	47.72

Brightness

The images' brightness is changed by adding an offset to all pixels. Then we clip the values between 0 and 1 again. Thanks to our normalization process for all images, the accuracy barely degrades.

When we add 0.2 to each pixel, the results are shown below.

```
Test pretrained model in: outputs/vgg13/default
load model: outputs/vgg13/default/best.pth
accuracy on test set = 92.87999826669693 percent

Test Accuracy of   airplane: 93% (934/1000)
Test Accuracy of automobile: 97% (974/1000)
Test Accuracy of       Bird: 90% (902/1000)
Test Accuracy of        Cat: 82% (827/1000)
Test Accuracy of       Deer: 94% (946/1000)
Test Accuracy of        Dog: 88% (884/1000)
Test Accuracy of       Frog: 95% (954/1000)
Test Accuracy of      Horse: 94% (947/1000)
Test Accuracy of       Ship: 95% (959/1000)
Test Accuracy of      Truck: 96% (961/1000)
```

Contrast

The images' brightness is changed by scaling all pixels. Then we clip the values between 0 and 1 again. Thanks to our normalization process for all images, the accuracy also stays high.

When we multiply 1.2 with each pixel value, the results are shown below.

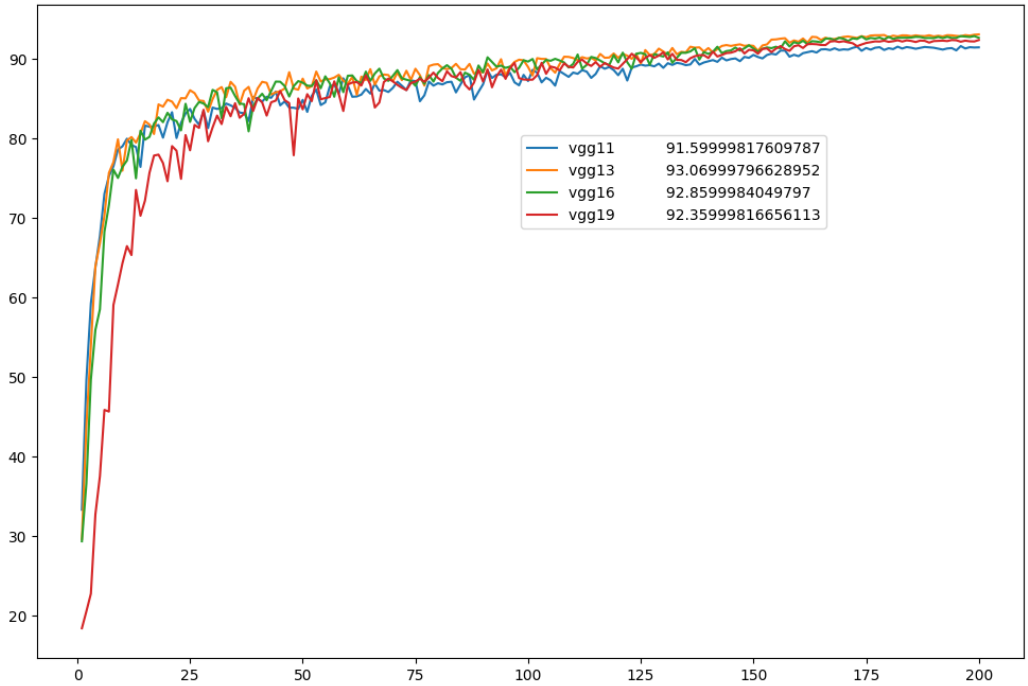
```
Test pretrained model in: outputs/vgg13/default
load model: outputs/vgg13/default/best.pth
accuracy on test set = 92.84999811649323 percent

Test Accuracy of   airplane: 94% (940/1000)
Test Accuracy of automobile: 97% (971/1000)
Test Accuracy of       Bird: 90% (902/1000)
Test Accuracy of        Cat: 83% (830/1000)
Test Accuracy of       Deer: 94% (943/1000)
Test Accuracy of        Dog: 88% (885/1000)
Test Accuracy of       Frog: 94% (948/1000)
Test Accuracy of      Horse: 95% (951/1000)
Test Accuracy of       Ship: 95% (959/1000)
Test Accuracy of      Truck: 95% (956/1000)
```

3.4 Ablation studies

3.4.1 Network

Convolutional Layer

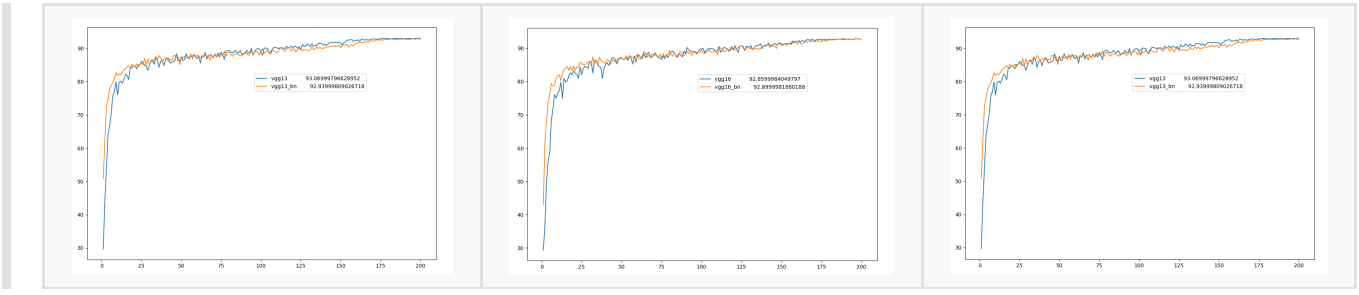


We use various VGG networks in this project, and their overall architectures are quite similar. What differs is their number of Convolutional Layers. So we can discover the role of Convolutional Layers from the different patterns they show. The accuracy of the test set concerning the training epochs of the 4 VGG classical models is listed above.

Recall our model structures illustrated in Part 2.2. To some extent, a proper number of Convolutional Layers can enhance the model's ability to extract features. However, if the features are not that hard to detect, the extra number of Convolutional layers go excess, which in turn may bring the networks into local optima. Especially, CIFAR-10 only has images of size 32 * 32, which means that excess convolutional layers will cause extra burden instead.

Batch Normalization

We train models both with batch normalization and without batch normalization.



Models with batch normalization converge faster than common ones. Besides, it reduces the importance of initial weights and hyperparameters, leading to more robustness.

The following chart shows our models' test accuracy rate with and w/o batch norm for VGG-13. The difference is very clear. Networks with batch norms are much more robust to the noises of inputs.

	Random noise(noise_num=200)	Sp noise(noise_num=205)	Gauss noise(var=0.02)
VGG-13 (w/o batch norm)/%	16.20	10.51	19.12
VGG-13 (with Batch norm)/%	48.58	42.95	47.59

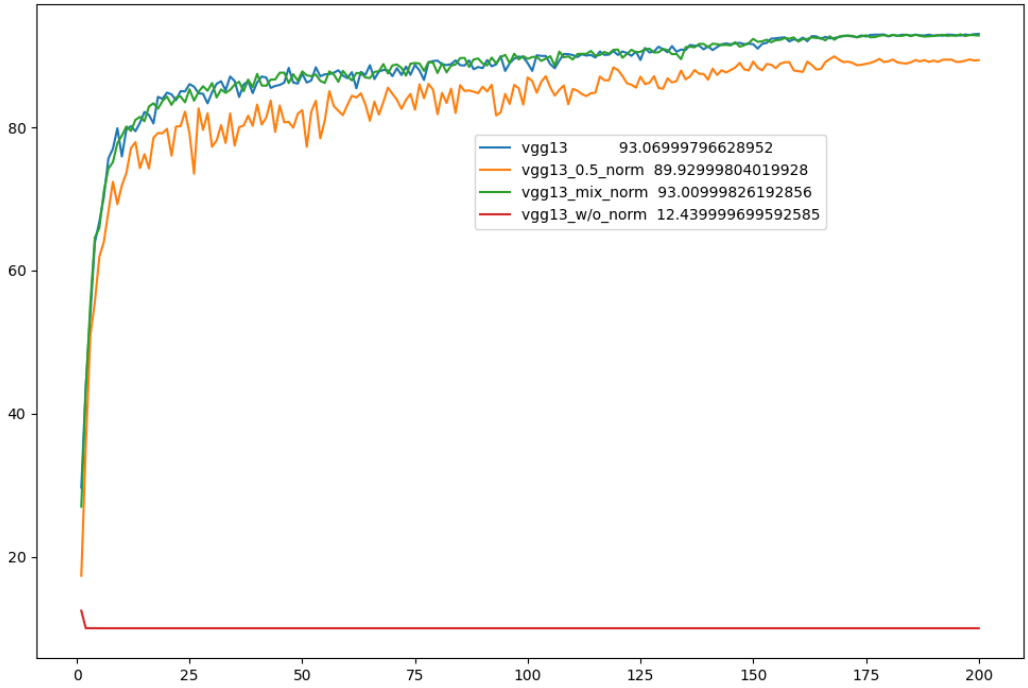
Their final results see no dramatic differences. Networks with batch normalization show lower accuracy than those without batch norm.

3.4.2 Datasets

Normalization

We normalize all the images before they are fed into the networks using their mean value and standard deviation on RGB channels respectively. This method proves effective when compared with other normalization methods or simply no normalization.

Specially, we compare our method with 3 other normalization strategies: no normalization, use 0.5 for both mean value and standard deviation, and normalizing the images' 3 channels together instead of separating channels.

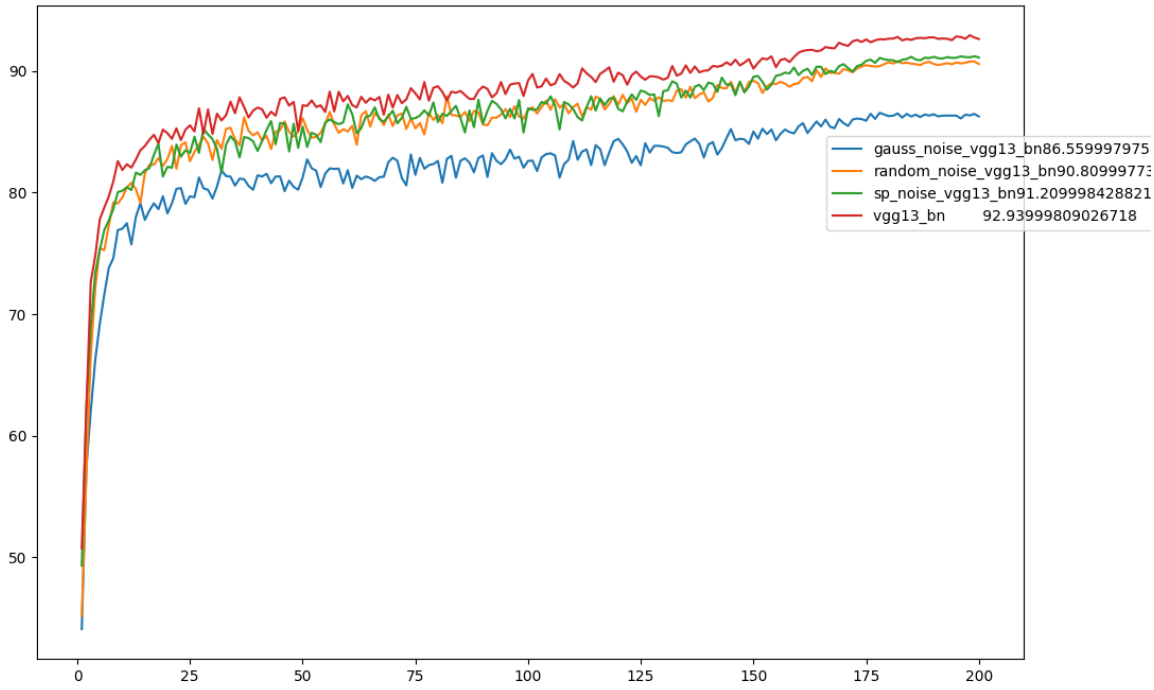


Without normalization, the model even cannot converge! Our normalization method beats other strategies in both convergence rate and best accuracy. The great difference in the *scale* of the numbers could cause problems when attempting to combine the values as features during modeling. Whereas normalization avoids these problems by creating new values that maintain the general distribution and ratios in the source data while keeping values within a scale applied across all numeric columns used in the models.

Train set with Noise

The models in Part 4.2 are all trained without noise images. Train set with noise is also utilized to test the what benefit the augmented data can bring to the models. To this end, we add various types of noises to the training set, and we train VGG-13 (Batch norm) models and then have them tested on both the original test sets and test set with noise.

The test accuracy on the **original** test set versus training epochs is shown below.



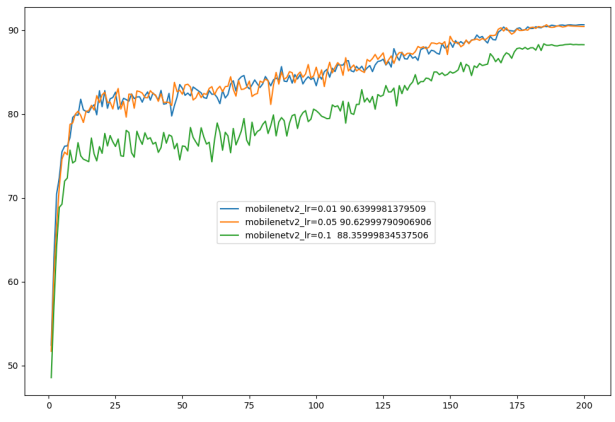
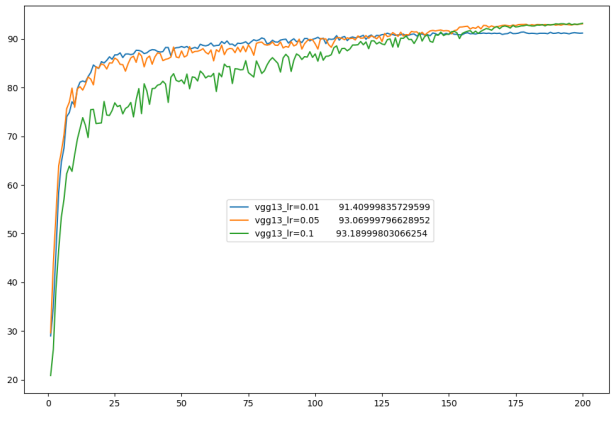
Although it costs some accuracy on the original test set, the robustness of the models is lifted a lot. The table below lists the accuracy on some noise datasets, and it takes VGG-13 (Batch norm) as an example.

Noise	Random noise(noise number=200)	Salt-and-Pepper noise(noise probability=0.1)	Gauss noise(variance=0.02)
Accuracy(%) with augmented train set	85.39	66.45	77.99
Accuracy(%) with original train set	48.96	42.23	47.72

3.4.3 Hyperparameters

Learning rate

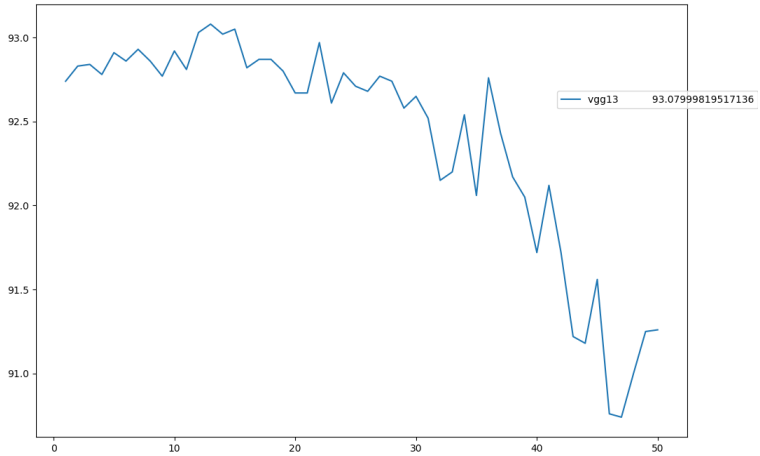
The most important hyperparameter is undoubtedly the learning rate. We modify our learning rate using the cosine annealing strategy, and the initial learning rate is set at 0.05 for all networks. To test the validity of the setting, we also experiment with other initial learning rates.

Mobilenetv2	VGG-13
	

Several classic initial learning rate are tested (including 0.01, 0.05, 0.1). 0.05 works best for most networks here, except for some situations where other learning rates have little edges over them.

Training epochs

We wonder if 200 epochs are enough to maximize a model's performance. To this end, we continually train VGG-13 for 50 epochs, we find that because of the extremely low loss and error rate on the train set, the accuracy barely improves. It drops overall.



4 Conclusion

We have presented various experiments with a classic network architecture based on a convolutional neural network. Carrying out ablation studies on all-around components of models, our project can methodically analyze the roles that each part of the network has. Our trained model achieves satisfactory results at very low costs with no extra data, which lays a solid foundation for our ability to further operate on more complicated models.